A Requirement-Driven Approach for Modelling Software Architectures

Yajna Kumar Makoondlall

A thesis submitted in partial fulfilment of the requirements of Kingston University for the degree of Doctor of Philosophy

Faculty of Science, Engineering and Computing
Kingston University

January 2020

ACKNOWLEDGEMENTS

Initially, I would like to express my deepest and most sincere gratitude for the invaluable support, continuous encouragement and scientific advice of **Prof. Dr. Souheil Khaddaj** during the course of this research. Thank you for believing in me when I had myself stopped believing in the project. The continuous encouragement and dedication of Professor Khaddaj is one of the main reasons why I managed to see the project to completion.

Next, I would like to thank my family. Thank you, mum and dad, for always being there for me. All great achievements are built on great foundations. You have ensured that I was able to complete my Bachelor's degree in the university of my choice, despite the tough financial conditions that would ensue. All the support throughout the years led me to the Phd course and an eventual submission. I would also like to thank my sister Toshima, my brother-in-law Vicky and their two children Girik and Kashika.

Next, I would like to thank my sister Manisha Makoondlall, who passed away in October 2016. The courage and tenacity with which you stood against the illness, which eventually terraced you, is second to none. The lesson to be learnt from you is simple: Fight until your last breath. If you are still alive, it means that the Lord of the Universe is not done with you. Therefore, live your life to the best of your abilities and fulfil your duties.

I would also like to thank Hema Bhakshi, who is always very accommodating and offers to house me when I visit the Kingston University. She was also very encouraging in the final year and offered to monitor my progress through weekly calls. Thank you once again.

Next, I would like to thank Dr. Bippin Makoond who suggested that I embark on this journey and proposed the initial idea for the research. Finally, I would like to thank the larger family, who are always there for support. Thank you to the Makoondlall, Makoond, Bikoo and Ramburuth family and countless others who have supported me and I cannot name all of you individually.

ABSTRACT

Throughout the software development lifecycle (SDLC) there are many pitfalls which software engineers have to face. Regardless of the methodology adopted, classic methodologies such as waterfall or more modern ones such as agile or scrum, defects can be injected in any phase of the SDLC. The main avenue to detect and remove defects is through Quality Assurance (QA) activities. The planned activities to detect, fix and remove defects occur later on and there is less effort spent in the initial phases of the SDLC to either detect, remove or prevent the injection of defects. In fact, the cost of detecting and fixing a defect in the later phases of the SDLC such as development, deployment, maintenance and support is much higher than detecting and fixing defects in the initial phases of the SDLC. The software architecture of the application also has an incidence on defect injection whereby the software architecture can be regarded as the fundamental structures of a software system. The impact of detecting and fixing defects later on is exacerbated for software architecture which are distributed, such as service-oriented architectures or micro-services.

Thus, the aim of this research is to develop a semi-automated framework to translate requirements into design with the aim of reducing the introduction of defects from the early phases of the SDLC. Part of the objectives of this work is to conceptualize a design for architectural paradigms such as object-oriented and service-oriented programming. The proposed framework uses a series of techniques from Natural Language Processing (NLP) and a blend of techniques from intelligent learning systems such as ontologies and neural networks to partially automate the translation of requirements into a design. The novelty focuses on moulding the design into an architecture which is better adapted for distributed systems. The framework is evaluated with a case study where the design and architecture from the framework is compared to a design and architecture which was drawn by a software architect. In addition, the evaluation using a case study aims to demonstrate the use of the framework and how each individual design and architecture artefacts fair.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

IT	Information Technology
QA	Quality Assurance
DP	Defect Prevention
SOA	Service Oriented Architecture
SDLC	Software Development Life Cycle
NLP	Natural Language Processing

CHAPTER 1: OVERVIEW AND BACKGROUND

1.1 Introduction

The task of overseeing an IT transformation project or a newly built software project to completion is a complicated one, which is full of pitfalls. Over the years there has been a series of failed IT or software projects [1][2], some of which have fast tracked the bankruptcy of the company concerned. The reasons why a software project fails vary hugely from project to project [4], but the results can be dire. Whilst it is true that software failures or rather software malfunction is likely to be around for as long software will be, all the key stakeholders involved in a project need to ensure that the project is completed with as little defects as possible. R. N. Charette [3] believes that more can be done to ensure the quality of software delivered. Apart from software quality it is important to deliver the projects on schedule and on budget. Some of the cancelled projects were simply called off as they exceeded their initial budget and were behind schedule. The projects became unsustainable and were simply abandoned.

In order to ensure that the delivered software systems do not contain defects and are not error prone, most companies use post implementation Testing and Quality Assurance models [5]. In a research carried out by Adeel et al. [6], the authors have handed out questionnaires to IT professionals in the industry with the aim of gathering information on their defect removal strategies and then investigate a series of Defect Prevention (DP) techniques. They noticed that System Testing was the most widely used technique (100%), followed by Integration testing (88%) and Unit testing (75%). Unit Testing is usually the testing carried out by the development team after a unit has been coded, Integration Testing can be regarded as the testing of several units to ensure that the units interact as expected and System Testing can be regarded as the testing of the System as a whole. In a typical waterfall methodology life cycle, testing is the last phase before a product is delivered.

If formal testing (Unit Testing, Integration Testing and System Testing) is the primary means of detecting defects, it means that defects will have to be fixed as a rework and a lot of regression testing will then be needed to make sure that the software deliverables are working as expected. This widely used approach addresses the defects after the solution has been designed and coded,

but the design defects and requirements defects are not necessarily addressed. The design used may not meet the requirements and other client needs (like scalability). The requirements used to design the solution may not meet all the client's exigencies as the requirements gathered may be incomplete, ambiguous and exclude tacit requirements.

1.2 Background information and challenges

The cost of fixing a defect in a production environment is much higher than in the initial phases of the Software Development Life Cycle (SDLC). Boehm and Basili [7] claimed that the cost of fixing a software defect in a production environment can be as high as 100 times the cost of fixing the same defect in the design or requirements phase. It is also important to note that they observe that the cost of fixing a software defect is closer to the 5:1 ratio for small noncritical software systems. However, researchers at the IBM Systems Science Institute state that the ratio is more likely to be 200 to 1 as shown in Figures 1.1.

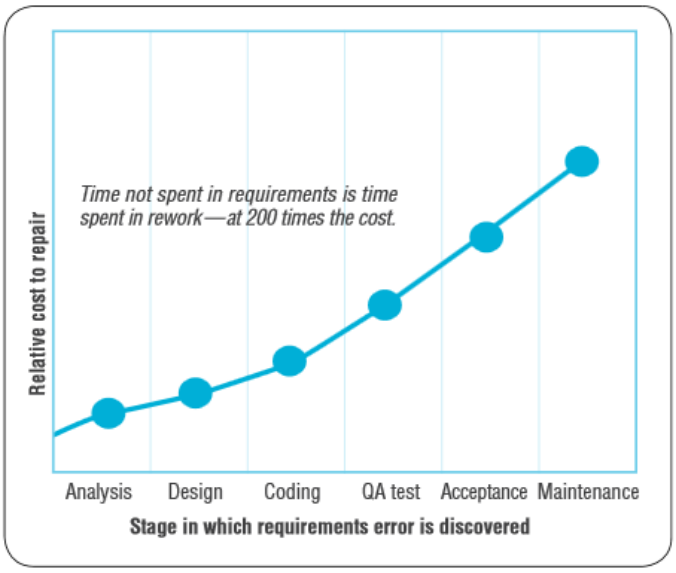


Figure 1.1 : Relative costs to fix software defects [8]

It is therefore worthwhile investigating techniques which may help to reduce defects from the earlier phases of the Software Development Lifecycle (SDLC) such as the requirement phase. Detecting defects in the requirement phase will mean that:

- The end product will match as closely as possible to the user's specification
- The architecture proposed is most likely going to satisfy the user's needs

- The piece of software is not developed to completion and then tested for defects. (where reworks are then the means of removing or eliminating defects.)

Software testing is not enough to remove all the defects and therefore other techniques also have to be used to eliminate defects. The situation is bad enough for monolithic systems, but it may be even worse for distributed systems, which are very often developed and implemented across different teams which span across different geographic locations.

In recent years, demand for service oriented distributed systems has been increasing rapidly with new technologies emerging to support the growth and diversity of users' requirements. Both Cloud Computing and Service Oriented Architecture (SOA) are so far the leading trends which businesses and companies are most likely going to adopt [9][10][11]. Thus, the current and future software architectures are more likely going to be distributed by nature, involving the interaction between diverse systems and platforms, resulting in systems which are more complex than monolithic systems, and require new defect detection mechanisms.

Thus, companies and software departments need to test and validate the pieces of software and architectures prior to deployment. Failure to do so early enough in the software development lifecycle (SDLC) can prove very costly. Appropriate verification and validation techniques need to be put in place so that there are as little defects as possible leaking into the latter phases of the SDLC. As more businesses and governments move towards distributed software systems such as SOA and Cloud Computing in order to maximize the capability of software solutions, it is necessary to devise strategies, methodologies and validation techniques to reduce the number of defects.

The defects found in distributed systems may occur in a service or out of the interaction amongst many services. The defects encountered in distributed systems are thus more difficult to fix than the defects encountered in monolithic systems [12]. As the modules in a distributed system start to interact with each other, there are often scenarios or states that have not been modelled upfront that start to emerge.

For example, let's consider a simple "Time and Pay" system, for a supermarket open seven days a week, as illustrated in Figure 1.3. The company has a Time system, which records the times at which employees clock in or clock out. There is then a different Pay (payroll) system that

calculates the salaries to be paid to the employees, based on the information supplied by the Time system. The important pieces of information are the number of hours worked, the pay type (Hourly employees paid at an hourly rate or salaried employee paid a fixed amount), the pay frequency (monthly, bi-weekly, semi-monthly, weekly) and other parameters to calculate the tax and National Insurance amounts to remit to the authorities. The Time system and the Pay system may have been tested individually and are given the green light from the testing and Quality Assurance (QA) teams. But the Time system may store data spanning over two days for an employee clocking in at Saturday 22 00 and leaving on Sunday 07 00. The Pay system may not be able to compute the employee's salary if the data spans over more than one day, especially if the hourly rate changes at midnight on Sunday. The employee may end up being underpaid for the hours worked.

Figure 1.2 : Simple (distributed) time and pay system

Thus, the defects from those emergent behaviours are harder to fix as they require more time and additional effort for rework analysis. To avoid the additional costs linked to software defects, it is worthwhile investigating techniques which ensure the pieces of software delivered are less defective. As outlined earlier, it can be far less expensive to eliminate the defects in the early phases of the SDLC, especially for distributed systems.

Therefore, this research aims to develop a framework methodology which is used to reduce the introduction of defects from the requirements phase and design phase, and therefore keeping the cost of rework lower and avoid using post implementation QA activities as the sole method to detect defects. The framework is used on natural language requirements which parses the requirements and uses a series of intelligent techniques to derive a design and an architecture from the requirements. The design and architecture are more adapted for the implementation of distributed systems, as they are abstracted, into modules which can easily be translated to a micro-

service architecture, service-oriented architecture or a distributed three-tier architecture. Figure 1.3 demonstrates the concepts and domain of the project as a whole.

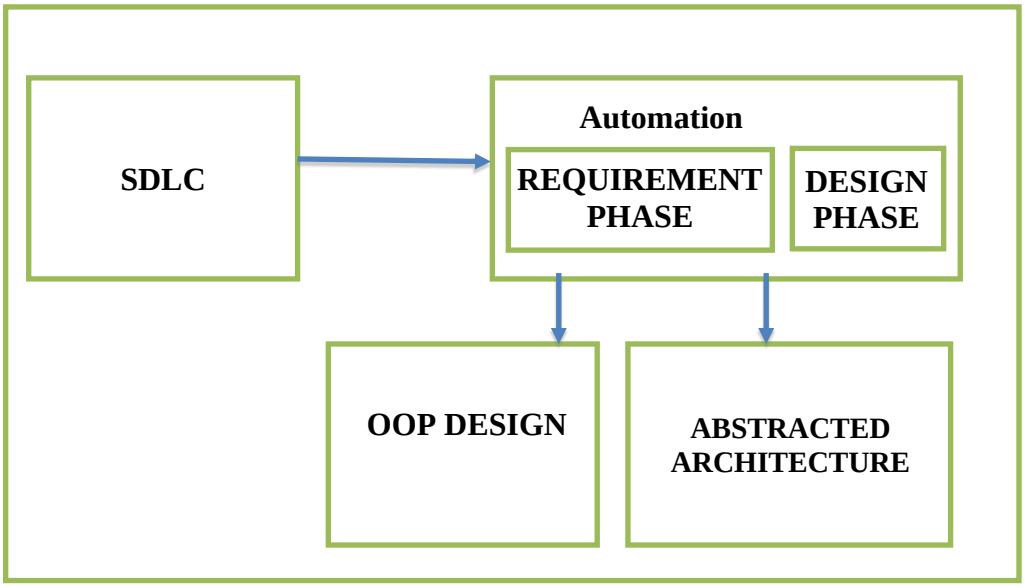


Figure 1.3 : The domain of the research

1.3 Aims and objectives

Defect prevention and reducing the introduction of defects are important activities in any software project. This research is aimed at creating a semi-automated approach to partially automate the translation of requirements into a design and an architecture for distributed systems. The key stages of the research are:

- Researching and documenting requirement attributes, which define the quality of requirements and enumerate manual requirement defect prevention techniques.
- Researching and listing automation techniques applied to the software development lifecycle to reduce defects.
- Researching, documenting and comparing existing approaches, which partially automate the translation of requirements into design artefacts and explaining the need for a new framework.

- Listing and explaining in details the components of the proposed framework. Applying the framework on a case study and comparing the results to a manual approach. The results are compared qualitatively and quantitatively.

1.4 Contributions to knowledge

The main contribution of this thesis is the development of a framework which can partially automate the translation of natural language requirements into design artefacts. Similar research has been published, but this research uses intelligent learning techniques such as an ontology and a neural network to abstract the design so that the final outcome is better adapted for architectures which are suitable for distributed systems.

The use of learning techniques within the framework allows for a more dynamic behaviour as the software can learn and grow when exposed to new data or can be configured to process new information differently. The final outcome comprises of multiple artefacts, including a use case diagram, a class diagram and three architectural design paradigms, namely a fine-grained architecture, a coarse-grained architecture and a thick-grained architecture. The fine-grained architecture can be implemented as a micro-service architecture, the medium-grained as a service-oriented architecture and the thick-grained as a three-tier architecture.

There are also two papers which were successfully published throughout the research work. They are listed below:

ZDLC: Towards Automated Software Construction and Migration
Y. K. Makoondlall, S. Khaddaj and B. Makoond, "ZDLC for the Early Stages of the Software Development Life Cycle," 2014 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, Xian Ning, 2014, pp. 6-12. https://ieeexplore.ieee.org/abstract/document/6999046

ZDLC: Towards Automated Software Construction and Migration
S. Khaddaj, Y. K. Makoondlall, B. Makoond, S. Chivukula and K. Keerthi, "ZDLC: Towards Automated Software Construction and Migration," 2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES), Guiyang, 2015, pp. 13-16. https://ieeexplore.ieee.org/abstract/document/7429545

The first paper entitled “ZDLC for the early stages of the Software Development Lifecycle” describes the initial vision of the research work. The paper covers the background literature and exposes the same argument to try and reduce defects from the earlier phases of the SDLC. Then a framework is proposed to model four categories of requirements for a design and architecture. The four categories of requirements were listed as data requirements, process requirements, communication requirements and non-functional requirements. As the research for the Phd continued, it was decided not to model the categories of the requirements but come up with an architecture for the whole of the requirements used as input to the framework.

I was the main author of that paper and the co-authors were Professor Souheil Khaddaj, the first supervisor, and Doctor Bippin Makoond, the contact person within Cognizant (ZDLC) collaborating with the Kingston University. The literature review of the paper is almost the same as presented in this thesis, except that it was condensed to fit a journal paper format. The paper exposes the initial design of the framework and how the framework could be part of the ZDLC suite of products. The paper was written after the initial monitoring report and it was an opportunity to test the validity of the research and check if the research was worthy of being published.

The second paper, entitled “ZDLC: Towards Automated Software Construction and Migration”, was published in 2015. Along with Professor Khaddaj, we were the main authors of the paper. The paper describes how the framework proposed could be modified so that it would cater for the migration of software codes from legacy systems to more modern software architectures. Whilst not directly exposing the work carried out in this thesis, the paper exposes how modifications could be brought to the framework so that it could be adapted to migrate from legacy systems to more modern architectures. In this case, legacy systems refer to classic monolithic architecture, the likes of which have been coded in COBOL, could be interpreted and an architecture proposed for the migration. The proposed architecture would be a distributed system, perhaps a three-tier architecture or service-oriented architecture, which is most likely going to be coded in a programming language like C# or Java. The aim is to reduce the risk of the migration phase.

1.5 Thesis organisation

This thesis starts with a brief overview of the relative cost of software malfunction in the software development life cycle (SDLC), and also explains why the situation may be worse for distributed systems and architectures. It also discusses why defect detection and prevention are much more important to be applied in the requirements and design phases. There is also a brief description of what the framework aims to achieve.

Chapter 2 is a literature review chapter which elaborates the definition of a defect and exposes how defects can seep into the various phases of the SDLC. There is a particular emphasis on requirement defects and how these have been categorised by other scholars. The chapter also lists a series of desirable requirement attributes and manual requirements defect detection techniques.

Chapter 3 is a literature review chapter which covers the various aspects of automation applied throughout the software development industry. The review covers the following areas where automation has been introduced: Code Generation, Configuration Management, Quality Assurance and Testing (Automation Testing), software building and DevOps. Later on, there is an overview of automation applied to the requirement and design phases, including how commercial tools have been used to manage requirements. The chapter then focuses on textual analysis techniques, used to either extract features from the requirements or automatically translate the requirements into a design. Finally, there is a critical analysis of all the existing techniques and the need for a new framework is explained.

Chapter 4 is a blend of literature review and a description of the framework. Firstly, a new framework is proposed and it is explained how the framework intends to change the initial phases of the SDLC. An overall description of the framework is provided and the associated workflow is described. Secondly, the individual components are explained and for each component the key pieces of research for that topic are covered. As each of the component is explained, there is a piece of pertinent literature accompanying the description, so that the theory and technology used for the framework can be understood. Finally, the way the framework aims to abstract the design into an architecture and the way the user is meant to use the learning system for user feedback are also explained.

Chapter 5 covers entirely original work as it provides a detailed walkthrough of each of the component of the framework. The implementation details, the logical rules and the technology used to implement each component are elaborated. For the NLP component, the transformation steps and the internal logic are explained and all the external libraries used to create this component are listed, along with a justification for the choice of that particular technology. For the design extraction component, the UML technology used and the rules used to extract design artefacts are listed. The XML technology and the technology used for the database are also enumerated. Next, the technology and implementation techniques used for the ontology and the neural networks are explained. Moreover, the external libraries used are listed, as well as the internal logic used to adapt the learning system for the framework. Finally, the implementation of the user feedback is provided.

Chapter 6 also covers original work and aims to demonstrate how the framework processes input requirements from a case study and evaluates the result of the framework when compared to a manual approach. The requirements are analysed for the manual elaboration of the design. Then the case study is loaded through the framework and there is a blow-by-blow description of how each of the steps in the framework transforms the input data. All the concepts obtained at each stage of the NLP component are listed and it is explained how these concepts are used by the design extraction. The setup, data and manner in which the design artefacts are enhanced by the ontology is explained. The output includes a clear list of attributes added to design by the ontology. The setup for the neural network and how it abstracts the design into an architecture are also explained. The outputs from the framework are first explained and then compared to the output generated from a manual approach. The similarity between the use cases and the class diagrams are compared. Then, the architectures are compared, starting with the three-tier architectures, followed by the coarse grain architectures, and finally concluding with the fine grain architectures. The similarity and differences between the manual approach and the framework are summarised.

Finally, **Chapter 7** concludes this thesis with suggestions for improvement in the framework and to incorporate further case studies to evaluate the framework as the experiment is conducted on a case study for a particular domain. In addition, suggestions are made to test the framework on other domains and case studies. Furthermore, the various aspects of the framework which could be improved were also listed.

CHAPTER 2: REQUIREMENT DEFECTS

2.1 Introduction

Before designing, implementing, deploying and using any software system, the team needs to be set on the purpose, the intended possible transactions and the motive for building the software system. It all starts with the requirements which are intended to capture all critical features as well as other aspects such as the aesthetics and the usability of the software system. During the requirements phase, the needs of the customer for the software system are collated by the business analysts in order to ensure that the software system would be fit for purpose when delivered. However, collecting all the intended requirements for a complex software system is not straight forward as there may be missed, incomplete or misleading requirements. All the shortcomings to capture the requirements adequately and precisely can lead to requirement defects.

Software defects can be introduced in any phase of the software development life cycle (requirements phase, design phase, development phase and implementation phase) and they affect adversely the quality of a product. Traditionally, software defects are detected and tracked by almost all well-established software project during dedicated phases to uncover defects. This information is then used to track the number of times service level agreements (SLA's) are breached, or how the software fails to deliver on the intended behaviour. These parameters, tracked over time, indicate the quality and reliability of a piece of software and can be used to track the quality of a software system in categories such as correctness, reliability, maintainability, performance, cost/benefit etc. Businesses spend a lot of time, money and effort to fix these defects as it affects the output of the software system.

Before analysing further requirement defects, it is important to understand the term "defect". Defects can be regarded as being of three types: error, fault and failure. The following definitions are quoted from [15] and are conform with the IEEE standards [13]:

- **Error** can be defined as a defect in the human thought process made while trying to understand given information, solve problems, or to use methods and tools. In the context of software requirements specifications, an error is a basic misconception of the actual needs of a user or customer.

- **Fault** – concrete manifestation of an error within the software. One error may cause several faults, and various errors may cause identical faults.
- **Failure** – departure of the operational software system behaviour from user expected requirements. A particular failure may be caused by several faults and some faults may never cause a failure.

Whilst it is true that defects would exist as long as software would exist, the aim is to come up with strategies that detect and address as many defects as possible. For every major software project, the requirements phase would be the first step towards getting started. The key stakeholders would lay out the outlines of the features which are desired and these are then translated into a set of requirements. The requirements can be vast and cover areas which go beyond features and include criteria such as scalability, ability to add features on the go, ability to distribute the same service on various platforms (web, tablet and mobile) and ability to derive data from the users' pattern of using the software. Once collated, all the requirements need to be interpreted and translated into a design which best matches the needs of the client as expressed by the requirements. In order to achieve a good design, the architect or team of architects need to be well versed with design trends, design patterns and the ability of a particular design to deliver on the requirements.

The task of translating the requirements into a design can be tricky as the requirements can be vague or deprived of meaning at the time that they are written. For example, customers may request that the software be "future proof" without really being sure what the "future proof" features are. Rarely are the requirements so vague, but very often the requirements may omit certain aspects that the client takes for granted. For example, when a customer requests for a piece of software that can be deployed on all platforms (web and mobile), the architect may assume that a responsive web design would be convenient for that customer. However, in the future the client may request for a mobile native app on the Apple app store or on the Google Play store, then the responsive web design may not be able to deliver on these requirements. All these scenarios are cited to help point out requirement defects, which are defects which seep into the requirement gathering stage and are carried into the design phase and later phases of the SDLC.

The strategy put in place by most companies to uncover defects, is to have a testing phase or a quality assurance process. The defects uncovered are then logged, resolved and kept in a defect

management system which usually makes use of a bug tracking tool (such as Bugzilla). Usually, the defect management system comprises of more elements than that and is integrated with a project management tool so that the metrics of the project can also be computed. For the defect management system to be efficient, the following key requirements are desired:

- A continual improvement of defect management system until the software / application is retired
- User and role-based allocation of issues/defects
- Defect identification and segregation (multiple products supported or multiple components within a product)
- Defect cross-referencing – preferably across segregation or components
- Root cause analysis (incomplete/inaccurate requirements, coding error, unit testing, system testing) [16].

The requirements of a software system determine how it is intended to be used, the key features needed so that it can be of value to the customer and how the users can use it to process service lines. The requirements therefore need to be clear, concise and purposeful so that it can fulfil on the service line for which it is intended for. However, just like all the phases of the SDLC are prone to defect injection, the requirements phase is also subject to defect injection. Section 2.2 covers an introduction to requirements defects and the relative cost of a requirement defect. Then section 2.3 covers a detailed description of requirement defects and how requirement defects are identified and classified. Section 2.4 covers a summary of desirable requirement attributes and their definitions. Section 2.5 covers defect detection and defect categories and strategies to avoid or reduce requirement defects. Finally, section 2.6 summarizes the chapter.

2.2 Defects in Software Development Life Cycle (SDLC)

In order for businesses to adapt to change, there is an increasing demand for dynamic and adaptive strategies to adjust to market pressures, particularly as they are progressively using more distributed software system and services. With the adoption of service-oriented architecture in distributed systems, there is a more acute need for less defect prone software systems and services. The use of distributed systems adds a layer of complexity as compared to monolithic systems as

these systems tend to span over several geographical locations with cultural differences and thereby increasing the pressure on business analysts, systems architects and developers to deliver cost-efficient software systems which are versatile. Overall market and environment pressures may lead to compromise(s) in software quality, which increases defect leakages, particularly from the early stages of the software development life cycle, as shown in figure 2.1.

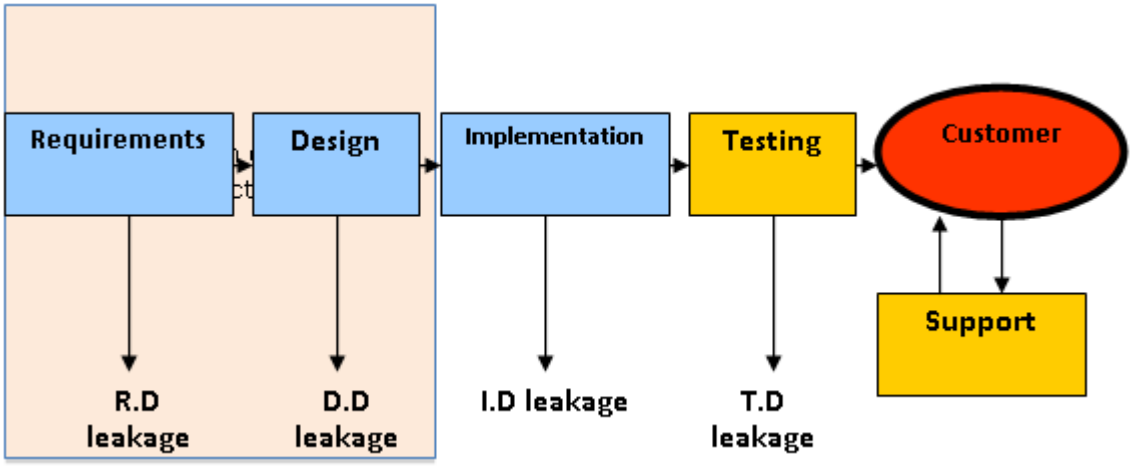


Figure 2.1: Defect leakages in SDLC

Defect identification, fixing and management remain important tasks in any software project and an organized problem-solving methodology is needed to identify, analyse, track, fix and prevent defects. The goal is to ensure that the end user encounters as few defects as possible and also that the software is performing as expected. Defect resolution is an iterative process of collecting data about the defect detected, performing a root cause analysis, implementing corrective measures to fix the defect and sharing the findings to prevent or reduce the same defects from occurring again. The first step towards defect detection and reduction starts with requirement analysis, which translate customer requirements into product specifications, while avoiding the introduction of defects in the process. The next step after that is elaborating the software architecture and further steps include the code review, quality assurance activities by testing the software system and logging all the defects observed so that they can be corrected and the last step is documentation of the software system, as illustrated in figure 2.2 [17].

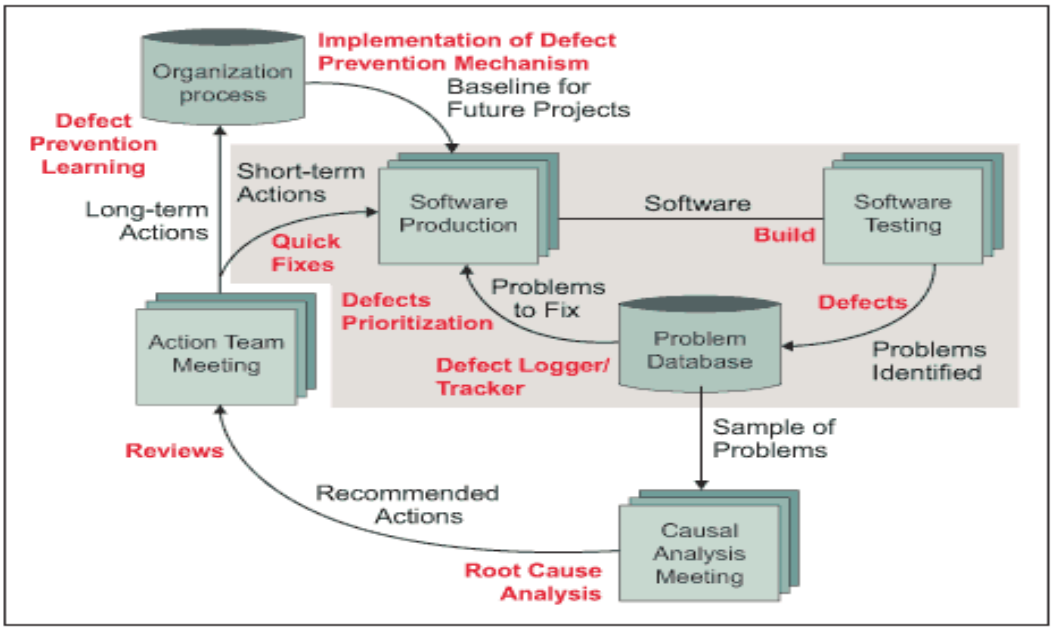


Figure 2.2: Defect prevention cycle [17]

The main advantage of early defect detection and prevention is that the cost of fixing a bug is significantly lower than what it would be if the same defect is detected much later in the software development life cycle. According to the National Institute of Standard Technology (NIST), the cost of fixing one bug found in the production stage of software is 15 hours compared to five hours of effort if the same bug was found in the coding stage [19]. This is backed by the Systems Sciences Institute at IBM, as they state that the cost to fix a defect realised after product release is four to five times as much as one recognised during design, and up to 100 times more than one realised in the maintenance phases as shown in figure 2.3, it is much cheaper to fix defects at early stages, such as requirement, design than at late stage of testing [17][18].

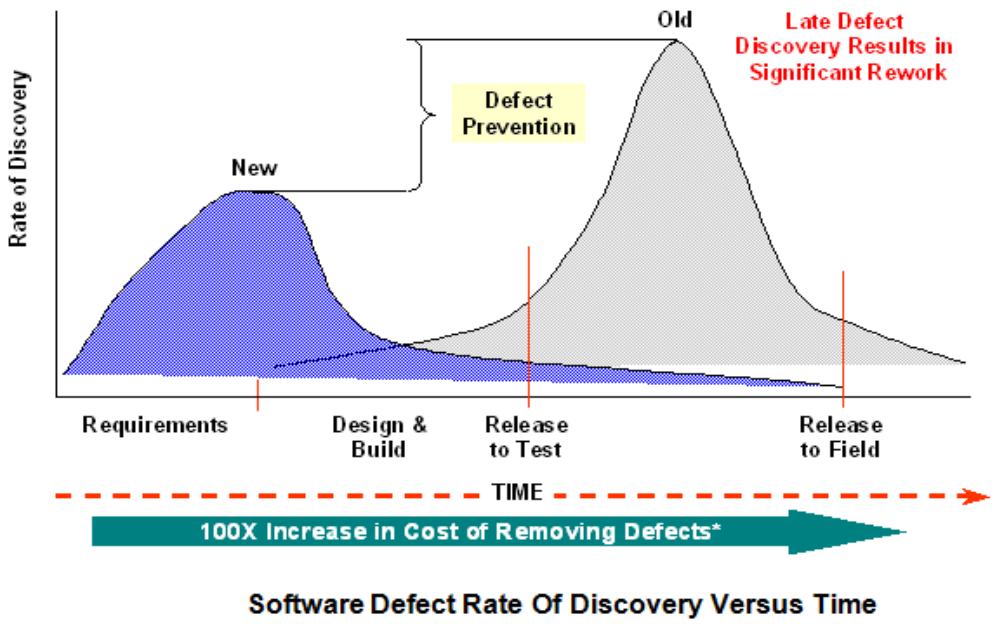


Figure 2.3: Software defect — rate of discovery v/s time [18]

The defect prevention process (DPP) is not itself a software development process but rather is a process to continually improve the development process. It originated in the software development environment and has been implemented mostly in software development organizations. DPP can be applied to any development process waterfall, prototyping, iterative, spiral, or others. As long as the defects are recorded, causal analysis can be performed and preventive actions mapped and implemented [20].

The middle of the waterfall process includes designing, coding, and testing. Figure 2.4 demonstrates a typical waterfall model for the prevention of the defects in software development life cycle. The aim of the software development process should be to produce high quality products and an improved design can contribute immensely towards that goal as a better design can prevent the majority of the errors. Figure 2.4 also depicts a linear-sequential life cycle model [21] and if followed rigorously, it can help to avoid most of the defects.

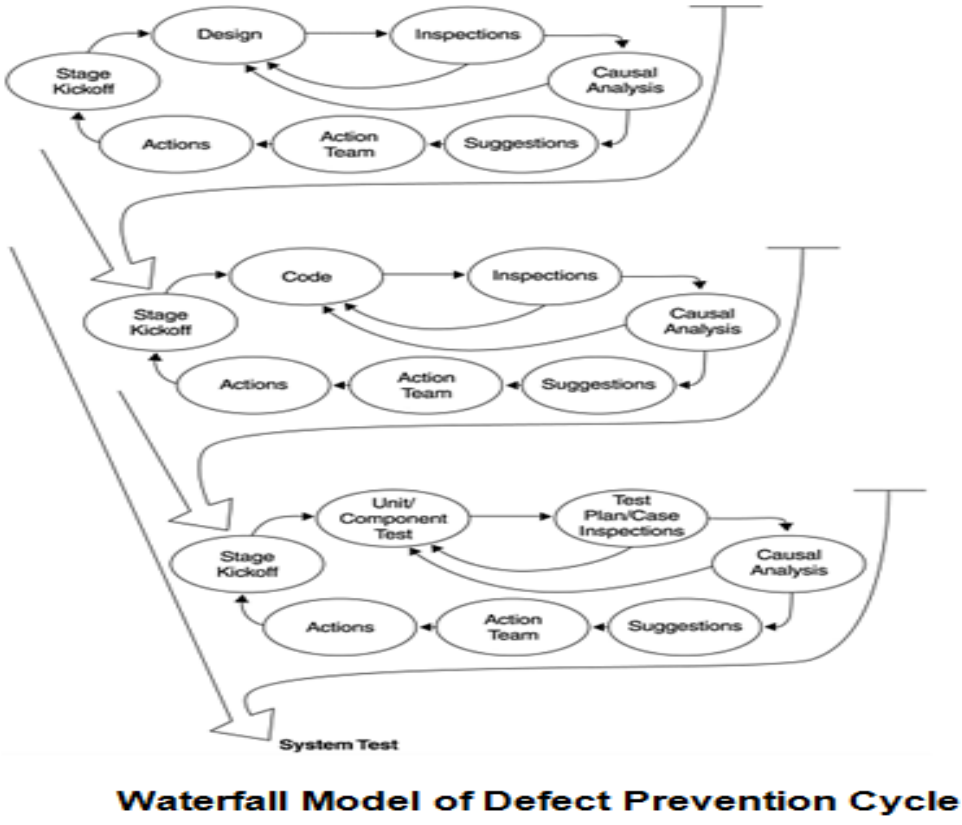


Figure 2.4: Waterfall model of defect prevention cycle [21]

Most software projects have scheduled activities to detect defects at different stages of the SDLC, and usually include activities such as design review, code review (code reviewed by a peer, also commonly known as peer-review), GUI review, functional testing, unit testing and user acceptance test (UAT). However, these activities are not always sufficient to identify, detect, prioritize and fix defects. As explained earlier, defects can be injected in any phases of the SDLC and the focus of this chapter is to analyse and define defects which are known as requirement defects.

2.3 Requirement defects

Software requirement defects occur when the software system fails to deliver on the needs of the stakeholder soliciting the software due to the fact that the requirements did not accurately capture the users' needs. There are multiple reasons why requirement defects occur and the rest of this chapter covers the different categories of defects and requirement defect detection strategies. There is also the process of Defect Prevention (DP) which aims at improving the quality of a piece of

software by identifying the common causes of defects, and changing the relevant process(es) so as to prevent that type of defect from recurring**.**

2.3.1 Defect identification

Requirement defects or anomalies can be regarded as requirements which have been erroneously captured or omitted from the requirement phase which would later lead to a defect. The requirement defect may not directly cause a software defect but may also do so indirectly. The defect might be caused by omitted requirements, or requirements which are incorrectly captured. For example, omitting the fact that a software system must allow for online payment, including digital wallets, may result to a platform which has an online payment system but which does not allow for payments from digital wallets. Other descriptions like the software "must be scalable" must be captured with more details; for example, how many users should be allowed to use the platform concurrently, or would the software be distributed on other platforms (like web, mobile, or desktop applications) in the future.

Software requirement defects can occur when the requirements are not understood and translated to a design that does not deliver on the customer's needs effectively and efficiently. For example, this may include interactions with existing pieces of software missing, system components do not communicate with each other, interaction with external software / hardware (such as database servers, cloud-based storage facilities or web services) and user interfaces or user input mechanism are missing. Reliable systems are often designed with the possibility of component failure in mind, so that the repercussions of these failures are mitigated in order to considerably reduce the odds of system failure. In classic fields of engineering, outside software engineering, the discipline of dependable system design is a well engrained practice.

Developing code free of defects is a major concern for the software development community. Requirement defects exist in a vast range of categories and types and the following sections analyse the different types of defects. Sorting and classifying defects are tedious and complex tasks because of the multiple points of view possible and the classification of defects can be subjective.

2.3.2 Defects classification

This section covers the work that has already been done on defect classification and reviews two taxonomies to classify defects and another approach which classifies requirements' errors. The

first taxonomy related to requirements defects, is the IEEE Std. 1028-1997 and it classifies the anomaly classes [13]. The second taxonomy is proposed by Margarido et al. [14] and it deals with the classification of defect types in requirement specification [14]. The last approach is proposed by Walia et al. [15] and it describes requirements' errors which lead to defects.

The IEEE Std. 1028-1997 categorized the anomaly classes as the following: missing, extra (superfluous), ambiguous, inconsistent, improvement desirable, not conforming to standards, risk-prone, factually incorrect, not-implementable and editorial [13]. Margarido et al. [14] classified defect types in requirements specifications by studying the literature review of existing defect taxonomies, deducing the adequacy of these classifiers to classify requirements defects, and identifying the frequency of the defect classifiers. The result of their work are the following requirement defects classes: missing or incomplete, incorrect information, inconsistent, ambiguous or unclear, misplaced, infeasible or non-verifiable, redundant or duplicate, typo or formatting, and not relevant or extraneous.

Walia et al. [15] developed a taxonomy of requirements' errors in the requirements phase, which lead to requirement defects. They recognised and composed fourteen sources of defects from literature survey of software engineering, psychology and human cognitive fields. They then categorized the errors into three high-level classes of requirements' errors: people errors, process errors, and documentation errors. Figure 2.5 demonstrates how the fourteen sources of defects are grouped under the three categories of errors.

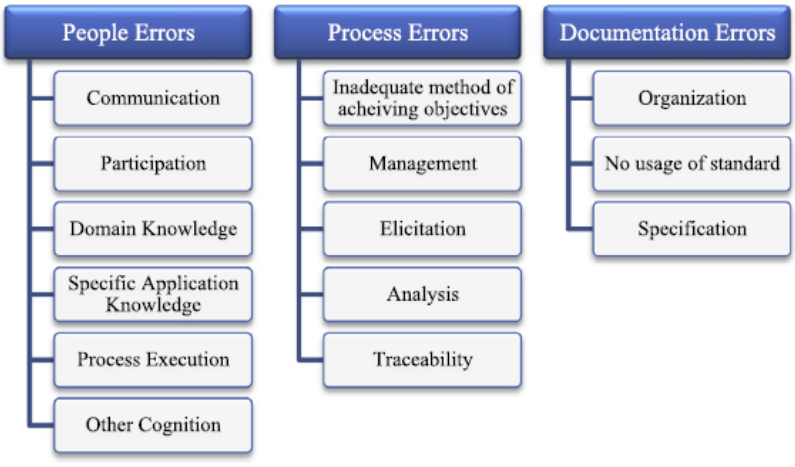


Figure 2.5: Classification of requirements' errors [15]

To better understand their approach, the main high-level categories of errors, leading to requirement defects are briefly explained below:

- People errors are the errors resulting from people involved in requirements preparation.
- Process errors are the errors that occur due to inadequate requirement engineering process, and selecting incorrect means of achieving goals and objectives.
- Documentation errors are the errors that occur due to improper organization and specification of requirements, regardless of whether the requirements author understood the requirements or not.

The key advantage of the requirement error taxonomy is that it covers errors that appear in the requirements engineering steps. These steps include requirements elicitation, requirements analysis and negotiation, requirements specification and requirements management which help in the requirements' verification. On the other hand, the main drawback is that this classification does not relate the types of defects to the causes leading to their appearances.

2.3.3 Requirement defects categories

There are various attempts to define and classify requirement defects. These different categories tend to group the requirement defects by their definition or how they occur. The following is a list of the requirement defects, which is quoted from the work of Alshazly et al. [22] and is consistent with the work of the three previously explored taxonomies [13-15]:

- **Omission** – Omission of requirements refer to fact that essential information related to the problem being solved by the software system has been omitted from requirements document or the requirements set is not complete.
- **Ambiguous** – Ambiguity means that the information present in the requirements document has more than one meaning and is subject to multiple interpretation.
- **Inconsistent** – Inconsistent requirements refer to the fact that one part of the requirements document describes a behaviour for a component and another part of the requirement document describes another behaviour of the same component. The SRS is not consistent throughout for the same component.

- **Superfluous** – Some of the information described in the requirements document is not relevant to the problem being solved and will not contribute to the final solution. Some of the information may be out of scope or beyond the mandate of the solution being proposed.
- **Incorrect** – Some of the information in the requirement documents contradicts information about the same component or contradicts relevant information in domain knowledge. The information in the requirement document may be in conflict with preceding documents or with industry standards.
- **Not conforming to standards** – Some of the information written in the requirement document does not conform to the standards determined by the quality assurance team members.
- **Not-implementable** – Some of the aspects of the requirements cannot be implemented because of human resource constraints, system constraints, budget, technology or other limitations.
- **Risk-prone** – Some requirements may be too risky as the resulting software would be unstable or cause the existing and interdependent software to be unstable. The requirements described pause the risk of being detrimental to other sub-systems or the whole software system.

When referring to requirement defects, the main category of defect that jumps to mind is ambiguity. Therefore, this paragraph explains "ambiguity" as a category of requirement defects in more detail. The requirements for a software project are written in Natural Languages (such as English, German, etc…) and these languages are inherently ambiguous [23]. Ambiguity arises from the fact that a sentence can be subject to several interpretations, leaving room for confusion and / or approximation. As a result, ambiguity very often leads to a requirement defect. Ambiguity is often specific to the language verbatim (linguistic ambiguity) but can come in different forms [23][24]. It is often thought that ambiguity can be of one of few types and it is rarely envisaged that there could be many types of ambiguity when it refers to natural language requirements. Figure 2.6 illustrates the various types of ambiguities which may arise in a natural language requirements document.

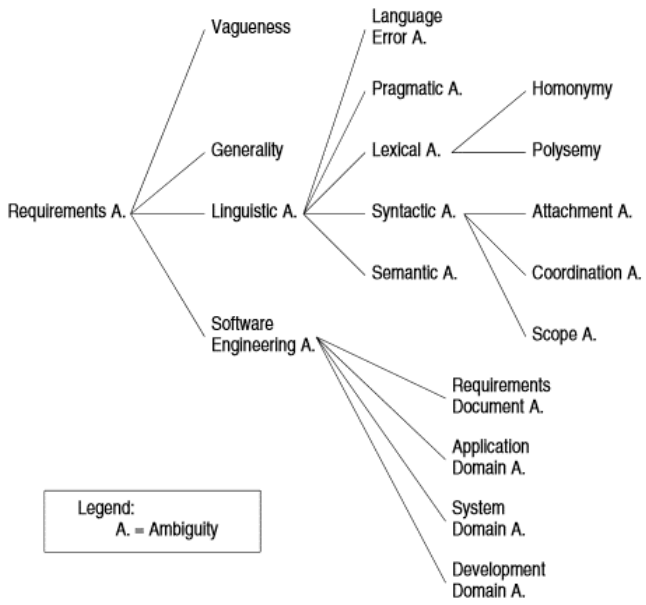


Figure 2.6: Taxonomy of ambiguity types [24]

In our case, we are mostly interested in the English language. Apart from the above-mentioned requirement defect categories, there are other aspects of the requirement engineering process to be considered. As explained by Walia et al. [15] (Figure 2.5), there are three types of errors which lead to requirement defects and they are people errors, process errors and documentation errors. People errors are errors caused by the resources involved in the process of capturing the requirement documents and writing the software requirement specification (SRS) document. Process errors are errors created when the requirement engineering process is conducted incorrectly and include requirement analysis errors. Finally, documentation errors comprise of errors injected during the process of writing the requirements regardless of the accuracy of the requirement engineering process or if the author actually understood the requirements or not. The work by Alshazly et al. [22] extends on the three categories of errors proposed by Walia et al. [15] and defines a few more categories of errors. They are:

- **Communication** errors occur when there is a miscommunication between various stakeholders involved in the requirement gathering process. For example, a business analyst from the software company and a client contact who is providing information on the exigencies of the software system may miss details as the client contact may not be technically savvy.

- **Participation** errors occur when there is insufficient or lack of participation from important stakeholders involved in the requirement gathering process.
- **Domain knowledge** errors occur when the author of the requirements lacks knowledge or experience about the problem domain.
- **Specification application knowledge** errors occur when the author of the requirement documents lack the knowledge about some aspects of the application commissioned or the problem domain.
- **Elicitation** errors occur when an inadequate elicitation process is used. For example, if all the requirements are gathered using a pre-determined questionnaire with little room to expand and explain the answers to the questions asked.
- **Analysis** errors occur after the requirements have been drafted and are analysed to take the project to the next stage.
- **Traceability** errors occur when there is no or little possibility to trace the source of the requirements which are passed from predecessors to successors and/or the absence of an effective change management process.
- **Organization** errors occur when the requirements are listed or drafted in an illogical fashion and arranged ineffectively in the requirements document, resulting in a poor organisation of the requirements document.
- **No-usage-of-documentation-standard** errors occur when the requirements are written in a way which does not conform to the documentation standards which have been set by the quality assurance representatives.
- **Specification** errors occur when described requirements are incorrectly specified, regardless of whether the technical team correctly understood the requirements. These errors include repetition of the same steps, typo errors and mistakes in naming and referencing the requirements.

Using these more expansive categories of errors and categories of requirement defects, Alshazly et al. [22] propose another taxonomy to classify and group requirement defects under categories of errors which lead to these errors in the requirement phase. Figure 2.7 illustrates their taxonomy:

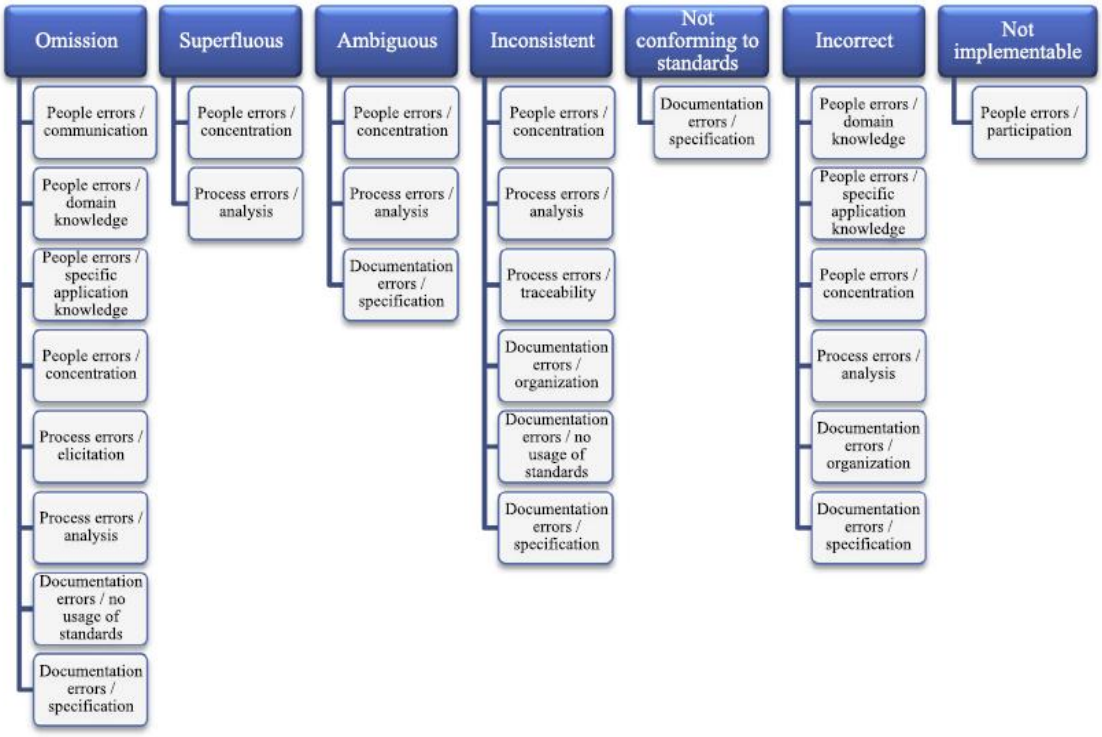


Figure 2.7: The taxonomy of defect types and their sources [22]

2.4 Requirement attributes

So far, the requirement defects, categories of defects and categories of errors leading to these defects have been briefly explained. Several taxonomies classifying defects and the errors leading to these defects have also been discussed. Whilst all this information walks through requirement defects, there is no information about the characteristics of the requirements which are desirable. This section provides a list of desirable requirement attributes which make the requirements less prone to defects.

Each requirement may have one or more attribute of the software that it describes. The following table lists the quality attributes of software specification and the associated definitions are presented [25][27]:

Requirement attribute	Definition
Complete	A complete requirements specification must precisely define all the real-world situations that would be exclusively encountered and the capacity of the software system to respond to them [26]. It must exclude situations that would not be encountered.
Consistent	A consistent specification is one where there is no conflict between individual requirement statements that define the behaviour of vital capabilities and these specified behavioural properties and limitations do not have an adverse impact on that behaviour [25]. The capability functions and performance level must be compatible and the required quality features (reliability, safety, security, etc.) must not contradict the capability's utility.
Correct	Correct requirements specification must accurately and precisely identify the unique conditions and restrictions of all situations that the desired capability will encounter. It must also define the capability's proper response to those situations [26][28]. The specification must define the desired capability's real-world operational environment, its interface to that environment and its interaction with that environment.
Modifiable	Requirements specifications can be said to be modifiable when related concerns are grouped together and unrelated concerns are separated [29][30]. This is achieved by a logical structuring of the requirements document. When structuring the document to be modifiable, the author must be careful that the ranking of the specifications according to stability and/or importance is not disturbed.
Ranked	Specification statements are best ranked according to stability and/or importance in the requirements document's organization and structure [25].

Testable	Testable specifications are specifications stated in such way that pass/fail or quantitative assessment criteria can be derived from the specification itself and/or referenced information [29][31]. It must be possible to derive test cases to evaluate the condition of the requirement (pass/fail).
Traceable	In order to achieve traceability, each statement of requirement must be uniquely identified [29]. This is achieved by the use of a consistent and logical scheme for assigning identification to each specification statement within the requirements document.
Unambiguous	An unambiguous requirement is a statement that can only be interpreted one way [29][32].
Valid	Requirements specifications are deemed valid when all the project participants, managers, customer representatives and engineers, are able to understand, analyse and accept or approve it [26]. Hence, the main reason why most specifications are expressed in natural language.
Verifiable	Verifiable requirement specifications are those which are at one level of abstraction consistent with those at another level of abstraction [26][31].

Table 2.1: Requirement attributes

2.5 Defect detection

Defect detection and prevention is an important part of software development as the quality, reliability and cost of the software product depends heavily on the software defect detection and prevention. In a software development life cycle, up to 40% of the project time can be dedicated to defect detection activities. Defects are found by activities specially planned and designed to unearth bugs and defects. These activities include design review, code inspection, GUI (graphical user interface) review, function and unit testing. Once the defects are identified, the defect management process should take over.

Software reviews and other requirement defect detection techniques have been extensively studied. This section provides a walk-through of all the techniques which are not automated or partially automated. Different techniques, frameworks and strategies, which use automation, focused on detecting and fixing requirement defects in software are presented in the next chapter. The next section lists the techniques, including reading techniques, and other strategies which can be used to reduce or avoid the requirement defects.

2.5.1 Defect detection strategies and techniques

The requirement validation techniques explained in this section are manual techniques which are dependent on human intervention. There is little or no automation in the techniques explained and they are used in order to detect and minimise requirements defects from creeping into the later phases of the SDLC. These techniques are:

- Requirements Reviews (Inspections)
- Requirements Review Reading Techniques
- Requirements prototyping
- Requirements Testing
- Viewpoint-oriented Requirements Validation

Each of the above-mentioned techniques are further expanded and explained in more details in the following sections.

2.5.1.1 Requirements reviews (Inspections)

The requirements review is very useful and effective validation technique to find out all major defects and it was presented by Fagan in 1976 [33]. In different literature most of evidences are in favour of this requirements validation technique. The inspection process can be very effective to eliminate undesired and ambiguous requirements from the SRS. During this process different sort of bugs can be detected at an early stage of software development, which would otherwise be detected at a later stage. Most companies, though they have limited resources, are recommended to prioritize the inspection of requirements over other artefacts like, for example, code inspections. In order to carry out an effective code review there are different steps needed such as planning,

overview, defects detection, defects collection and defects correction. These steps are briefly explained in the following paragraphs.

During the planning step, multiple meetings are organized in order to categorize the material to be inspected. Before the material can be presented for inspection, the software requirement specifications (SRS) need to be complete and free from unwanted and ambiguous requirements. Different people are also selected, assigned different roles and given material to review [34].

In the overview step, the author of the SRS presents it to the inspection team members and explains it in details. The purpose of this step is to ensure that the SRS is readable and understandable to the inspection team members, but has the disadvantages of consuming time and resources and a consensus may be hard to reach amongst inspection team members [35].

In the defect detection phase, the aim is to detect all possible defects from the SRS and there are two ways to go about it. The defects can be detected individually or different teams can be formed to detect the defects. Alternatively, this phase can also be divided in mini phases which are known as Phase Inspection Method [34][35]. The inspection is broken down into smaller sections and two or more inspectors are assigned for each phase. After a complete cycle of inspection, all the bugs can be collated and is more effective at removing defects from the SRS, but is also costlier.

In the defect detection step, the critical steps are:

- All defects found by the inspectors, from the SRS are collected and documented properly.
- Decide if each of the defect raised is really a defect or not.
- Decide if the infected modules need to be re-inspected.

Multiple meetings are then held to review the defects and the meetings include several resources which may include an inspector, a moderator and a writer. The process can be iterative where the inspectors go through the SRS (or portions) of the SRS multiple times. In the end, the list of defects found is consolidated.

Finally, there is the defect correction where the author of the SRS is sent the final list of defects from the previous steps. The latter is expected to correct the SRS so as to eliminate all the defects raised by the inspectors.

2.5.1.2 Requirements review reading techniques

Defect detection activities can occur in various manners and this section outlines some of the categories created for defect detection. The focus is on requirements defects and the defect detection categories focus on the categories of defect detection set up to detect requirement defects. Some of these categories include:

- Scenario based reading techniques.
    - Perspective based reading techniques.
    - Defect based reading techniques.
- Checklist based reading techniques.
- Usage based reading techniques.
- Ad-hoc reading technique.

There are various studies carried out trying to compare the differences between the reading techniques and the results and efficiency between these techniques [36-42]. The following paragraphs provide a brief explanation of each technique, without engaging into any comparative analysis between the different reading techniques.

Scenario based reading technique is an approach whereby different reviewers have the arduous responsibilities and are guided in their reading by specific scenarios which aim at constructing a model, instead of just passive reading. In this case, a scenario denotes a script or procedure that the reviewer should follow. There are two variants of scenario-based reading techniques and they are: Defect-Based Reading [46] and Perspective-Based Reading [47]. The former (subsequently denoted by DBR) concentrates on specific defect classes, while the latter (subsequently denoted by PBR) focuses on the points of view of the users of a document. Another part of the inspection process is the compilation of defects into a consolidated defect list where all individual reviewers' defect lists are combined. This step may include the removal of false positives (reported defects that were not considered to be actual defects) as well as the detection of new defects.

Perspective-Based-Reading (PBR) techniques focuses on the point of view or needs of the customer or target audience of the requirement document. The technique examines artefacts from

different perspectives of the users of software documents so as to improve efficiency by minimizing the overlap among the faults found by the reviewers [42]. When using PBR techniques, the people who have a role in the software development process or the maintenance process are identified. Once the roles (unique perspective) are identified, a list of questions are prepared for that stakeholder and an inspector reads the SRS by trying to answer the questions for that role. There may be multiple scenarios for one perspective. The focus is to understand the software product from each perspective, one at a time so that the inspector for that role (perspective) can identify the defects [43] pertinent to that perspective.

Defect-Based-Reading (DBR) technique concentrates on specific defect types in requirements, where requirements are expressed by state machine notation called Software Cost Reduction (SCR) [45]. The specific classes which are usually focused on include data type inconsistency, incorrect functions, and ambiguity or missing information. The main steps for executing the DBR technique are listed below [44]:

1. Defects are classified.

2. A set of questions is posed for each defect class.

3. Scenarios which are derived from checklists are built.

4. Each scenario is assigned to a reviewer to detect a particular type of defects.

The main advantage of this technique is that it provides guidance to the inspector about ‘‘What and How’’ needs to be inspected for [38]. On the other hand, the main disadvantage is that the scenarios limit the attention of the inspector to the detection of particular defects that are defined by the custom guidance [43]. Therefore, other types of defects may not be detected unlike the checklist technique which searches for any defects in the requirement documents [48].

In **Checklist-Based-Reading** (CBR) technique, the reviewer goes through a checklist which contains a list of questions or statements which are designed to find particular categories of faults [43]. The reviewer goes through a list of questions which are designed to trigger a binary answer (yes/no answers). The questions are based on previous knowledge of typical defects [37] and the checklist should be designed so that it does not exceed a page for each document [44]. The CBR technique does not provide instructions on how the inspection can be performed and is thus

considered as a non-systematic technique [40]. Using this technique, the reader is responsible for all the inspection processes and finding all possible defects.

In general, reading techniques are devised to detect defect in the SRS, regardless of the impact of the defect on the software. **Usage based reading technique** (UBR) is designed to detect defects that have the most negative impact on the software system. The key assumption is that all defects are not the same and the technique is guided by a prioritized, requirements level use case model during the individual preparation of the software inspection. Therefore, the requirements are broken down into use cases which are then prioritised so that the defects likely to impact the most critical use cases are known prior to inspection. UBR focuses on the users' needs as the review of the inspectors is performed with a list of developed and prioritised use cases, ranked in order of impact and priority for the software system commissioned.

The **Ad-hoc Reading Technique** works in ad-hoc manner as there is no proper mechanism to review the requirements. All members inspecting the requirements, do so looking for unwanted requirements from requirements specification, without any proper direction or guidelines [41]. This technique is not recommended for non-experienced members and the defect correction is made on the bases of experience of inspectors [34].

2.5.1.3 Requirements prototyping

In requirements prototyping, a prototype of the requested software is designed to understand the customer's needs. A well-designed prototype can help to understand the behaviour of the requested software and reduce integration work with existing software. For requirements prototyping to be efficient the software engineers need to work closely with the customers in order to gather the appropriate requirements. After the requirements are gathered, the software engineers can start building the prototype and it also means that the customers are part of the requirements validation process. By working closely with the team developing the prototype, the customer can better gauge the behaviour of the requested software and the process also makes it easier for the customer to explain all of the requirements to the team of software engineers.

There are two main types of prototypes and they are evolutionary prototyping and throw-away prototyping. In evolutionary prototyping, the customer works closely with the team of software engineers and the prototype development matches the customer needs as much as possible. The

prototype then needs to be changed, with an agile strategy [49] so that the prototype can evolve to integrate with existing software or be refined to better match the customer's needs. With throw-away prototyping, the prototype is not integrated with the final system or is not part of the final solution. The aim of building the prototype is for the customer and the developers to agree on the requirements and use the prototype to clarify any doubt and ensure that the requirements and the software commissioned would match the customer's needs [49]. The prototype is then discarded after the requirements are finalized and does not constitute a part of the final software to be built.

2.5.1.4 Requirements testing.

In a typical software development life cycle the software and hence the requirements for that software, are tested after development and before the software is released to the customer. With requirement testing, the requirements of the software requirements specification (SRS) are tested prior to the development phase [50]. In order to test the requirements, test cases for the requirements are designed from against the requirements. Inconsistent, incomplete and ambiguous requirements would make this task harder, but at the same time, this approach helps to identify these types of faulty requirements prior to development. The test cases are then executed at the end of the development phase during the final quality assurance activities. If done correctly, requirements testing can reduce the workload on the final quality assurance activities [50].

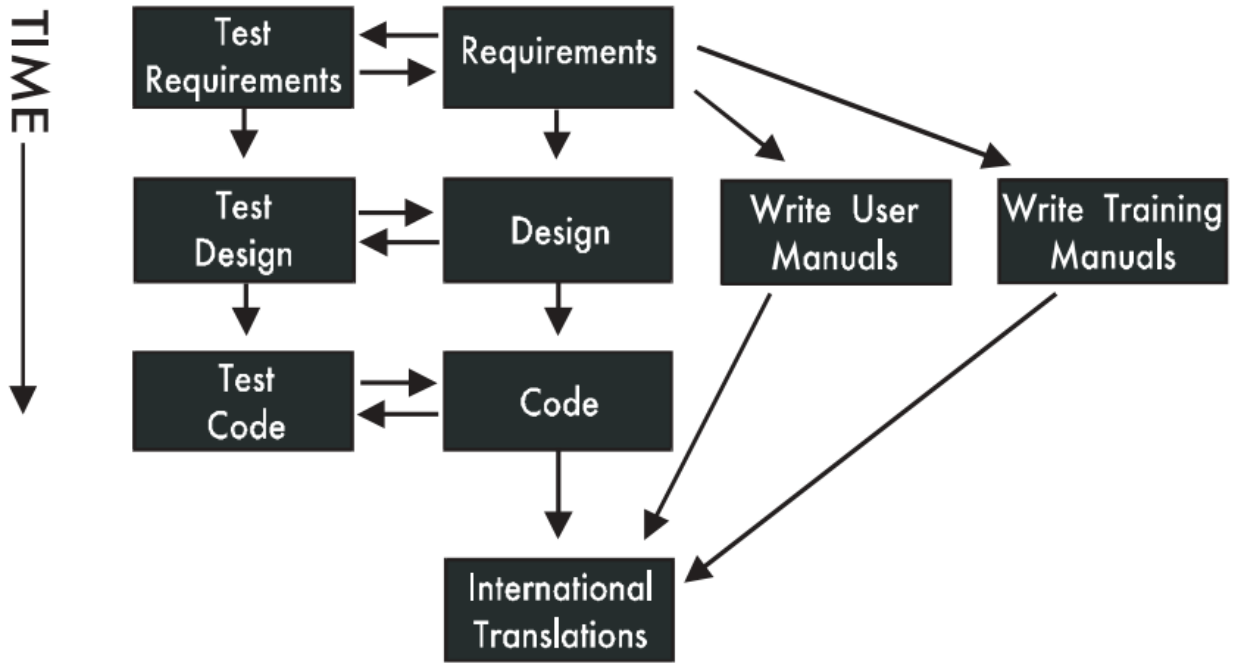


Figure 2.8: Lifecycle with testable requirements and integrated testing [51]

2.5.2 Defect prevention activities

Apart from reading techniques and defect detection techniques in the requirements phase, there are other defect prevention techniques which have been developed to reduce the injection of defects throughout the software development lifecycle.

The five general activities of defect prevention are [17]:

1. Software Requirements Analysis

Errors in software requirements and software design documents are more frequent than errors in the source code itself, according to *Computer Finance Magazine* [52]. Defects introduced during the requirements and design phase are not only more likely but also are more tedious and harder to remove. Front-end errors in requirements and design cannot be found and removed via testing, but instead need pre-test reviews and inspections.

2. Reviews: Self-Review and Peer Review

Self-review is one of the most effective activities in uncovering the defects as it helps to discover defects which may later be discovered by a testing team or directly by the customer. The majority of the software organizations is now integrating a review phase as part of "coding best practices" and in order to increase their product quality and reduce injection of defects.

3. Defect Logging and Documentation

Effective defect tracking begins with a systematic process. A structured tracking process begins with initially logging the defects, investigating the defects, then providing the structure to resolve them. Defect analysis and reporting offer a powerful means to manage defects and defect depletion trends, hence, costs. This should be part of a well thought of quality assurance strategy which ensures that the defects are captured, logged and tracked on time. The quality assurance (QA) process should also carry out recursive analysis (like a root cause analysis) so that the lessons learned from the causes of the defects are fed back to the key stakeholders and the same defects are not repeated over time.

4. Root Cause Analysis and Preventive Measures Determination

After the defects are logged and documented, they then need to be analysed so that factors like the root cause of the defect can be found. Generally, the designated defect prevention coordinator or development project leader or quality assurance manager organizes a meeting to explore root causes of the defects. Root cause analysis is the process of finding and eliminating the cause, of the defect so as to prevent the same defect from recurring or to reduce the likelihood of the same defect being created under similar circumstances. Finding the causes and eliminating them are equally important. The cause-and-effect diagram, also known as a fishbone diagram, is a simple graphical technique for sorting and relating factors that contribute to a given situation.

Once the root causes are documented, it is incumbent upon the team to find ways to eliminate or reduce the root causes of the defects. There is usually a brainstorming session, the objective of which is to determine what changes should be incorporated in the processes so that recurrence of the defects can be minimized.

5. Embedding Procedures into Software Development Process

Implementation is the most challenging phase of the SDLC to include defect prevention activities. It requires total commitment from both the development team and management. A plan of action is made for deployment of the modification of the existing processes or introduction of the new code or modules with the consent of management and the team. Monthly status of the team should mention the severity of the defects and their analysis.

2.6 Summary

Requirement gathering is a very important part of the SDLC and the quality of the requirements affect the rest of the SDLC and the software developed. There are various key factors to consider when designing the architecture of a piece of software and there is a heavy dependency on the requirements phase. The architect would propose an architecture which is thought to match the customer's needs best and can deliver on the requests of the customer. From past studies by the software development communities, it was shown that errors introduced in the requirements and design phase are the costliest and most detrimental to the final software. The detection of requirement defects is primordial to ensure quality software delivered, without a significant amount of money spent on correction of defects. This chapter provided an insight of the relative cost of defects in the SDLC, an overview of requirement defects, requirement defects identification

and classification, techniques to reduce and prevent requirement defects. Whilst the reading techniques can be effective, they are manual approaches to detect and remove requirement defects and they can be tedious and time-consuming.

Defect management is an important task to reduce the injection of defects in the SDLC and is an ongoing process of collecting defect data, performing root cause analysis, determining and implementing the corrective actions and sharing the findings learned to avoid future defects. Requirements defects are defects which are injected due to a deficiency in the requirement gathering process or the system requirement specification (SRS) document. They can be classified as omitted requirements, ambiguity, inconsistency, superfluous requirements, incorrect requirements, not conforming to standards, not implementable and risk prone. The quality, reliability and cost of a software depend heavily of the quality of the requirements and the ability of preventing requirement defects from being injected into the SDLC. The main defect detection strategies are requirements reviews (inspections), requirements review reading techniques, requirements prototyping, requirements testing and viewpoint-oriented requirements validation. Some of the defect prevention strategies explained are mostly a team effort as opposed to an individual exercise. The software development team should endeavour to identify and detect as early as possible so as to reduce resolution time and project costs. Most of the strategies and techniques discussed in this chapter are manual approaches and are not automated. These tend to be time-consuming, tedious and depends on the experience of the reviewer to unearth defects. In order to accelerate the process of detecting defects, some researchers have explored the possibility of using automation or semi-automated techniques to detect and process defects from the requirements and design phases at a faster pace. The next chapter provides a walk-through of these approaches which are automated or semi-automated in order to detect requirement defects, and help with the elaboration of a design. The goal of these approaches is to reduce the manual work needed for the detection of defects, reduce the injection of defects in the requirements and design phases and also process a larger volume of requirements at a faster pace, through the use of automation.

CHAPTER 3: AUTOMATION OF SOFTWARE DEVELOPMENT AND REQUIREMENT PHASE

3.1 Introduction

Creating large scale complex software from scratch or even with existing codes or prototypes is not a simple undertaking and, as elaborated in the previous chapter, is prone to defect injection at almost any stage. Since defects can creep into the software development lifecycle at any stage, it is important to find strategies, techniques and tools, which can help to minimize the introduction of defects. In that respect, automation has been used by various stakeholders to navigate the mine field of creating software with minimal defect injection. Automation can have many different interpretations as it can refer to any accelerator, tool or framework, which can accelerate or reduce manual work or manual intervention from the user to complete a task. The aim of automation can also be to guarantee repeatable and reliable results, which may be missed by manual execution, especially if the tasks are mundane and repetitive but still have a certain degree of variability.

Automation has been introduced in almost all the sectors of the economy and software engineering has not been spared. Whilst it is true that software engineering mostly does not involve physical machinery, some of the virtual tasks required to create a large piece of software system have been automated. Almost all the aspects of software engineering have been automated or partially automated, including requirement engineering, design to code generation and software automation testing. The aim of automation is to accelerate known steps, which are either too tedious, time-consuming or repetitive so that the user spends less time on mundane and less productive tasks. The gain of such automation is to ensure that the results are consistent, reliable and repeatable while also freeing up time for resources to focus on innovative, differentiating and value-added tasks. Since most of the tasks in software engineering do not require physical tasks and qualified professionals (like a certified engineer or a lawyer) to be carried out, it may be difficult to conceive how automation can be introduced. The focus of this chapter is to explain how automation has

been introduced in software engineering with the aim of improving accuracy, reliability and reduce processing time for certain tasks.

This chapter covers an introduction to automation and how it is used in all the phases of the software development life cycle. Then section 3.2 focuses on the use of automation in the later phases of the SDLC. Section 3.3 covers a series of the techniques used in requirement engineering and the design phase. Section 3.4 covers the textual analysis tools and frameworks to translate requirements into a design. Section 3.5 covers a comparative analysis of the tools and techniques and finally section 3.6 summarizes the chapter.

3.2 Automation in the SDLC

There are several phases in the SDLC and the five main ones are requirement gathering, design (and architecture), implementation (development and coding), quality assurance and deployment. There are other phases which may be added, like support activities which happen after the deployment and also the above-mentioned phases may be further sub-divided into other phases or sub-phases. In our case, we shall consider the five main phases of the SDLC mentioned above.

Initially the software industry started with tools or common artefacts to understand, explain and share software design so that it would be easy and uniform to represent and depict software design and increase collaboration amongst groups of software engineers. This kind of thinking led to the creation of the UML (Unified Modelling Language) and the OMG (Object Management group) so that the design of software can be expressed following common grounds. These kinds of uniformity in design led to the standardization of software design and teams also did not have to invent their own graphical representation of a piece of software in order to explain the design. In late 2000's, the MDA (Model-Driven Architecture) initiative was launched by Object Management Group (OMG) to promote the use of design models as the essential artefacts of software development. The MDA initiative was followed by the MDD (ModelDriven Development) or MDE2 (Model-Driven Engineering), which led to a new paradigm in software development where models are the primary software artefacts and transformations are the primary operations on models. In MDE, a lot of consideration is needed to ensure that the quality of models generates the right artefacts. In addition, since defects can be detected earlier and corrected in models, this led to an improvement in the quality of the models and would ultimately reduce

maintenance costs [54]. The use of design models leads to "model-based software development" whereby the models accepted as part of the design drive the code. For example, if a class is defined in a class diagram, a lot of the code for that class can be derived like all the "setters" and "getters" method, the attributes and the name and expected behaviour of the methods. Figure 3.1 shows the 6c Model (completeness, correctness, confinement, changeability, consistency and comprehensibility) are used to ensure that a model can be produced with minimal disruption.

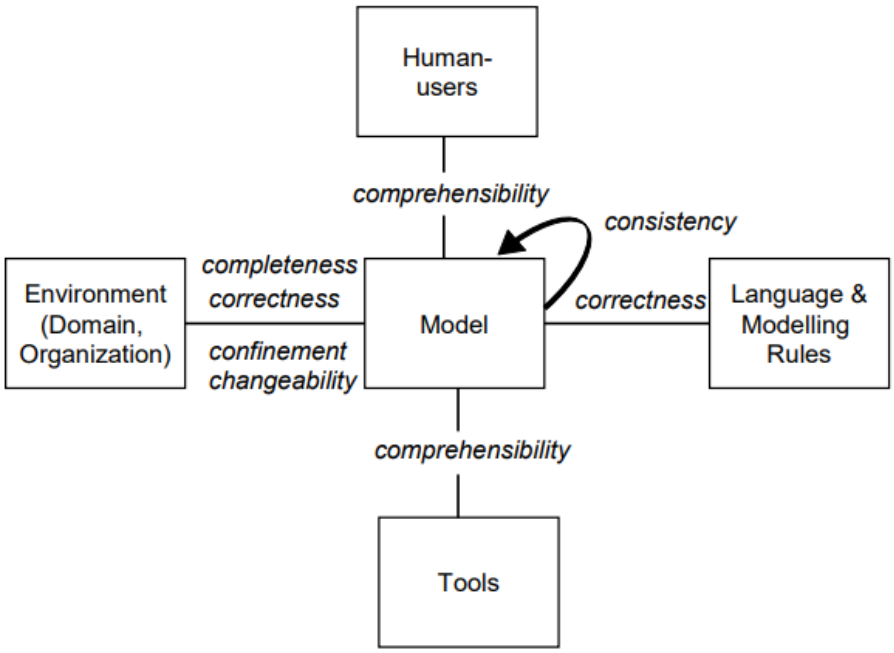


Figure 3.1: The 6c model quality goals [55]

Another variant of the 6C Model shows how it can be used to ensure that the design is "correct" (correctness) with the modeller using rules, guidelines, tools and modelling language to deliver a design. Other human users are solicited to ensure comprehensibility and deliver code. The model is consistent and analysis and generation tools are used to ensure comprehensibility and code generation. The model is also kept in check (ensuring completeness, correctness, confinement and changeability) with the application to real world domain and organizations and it is shown in figure 3.2.

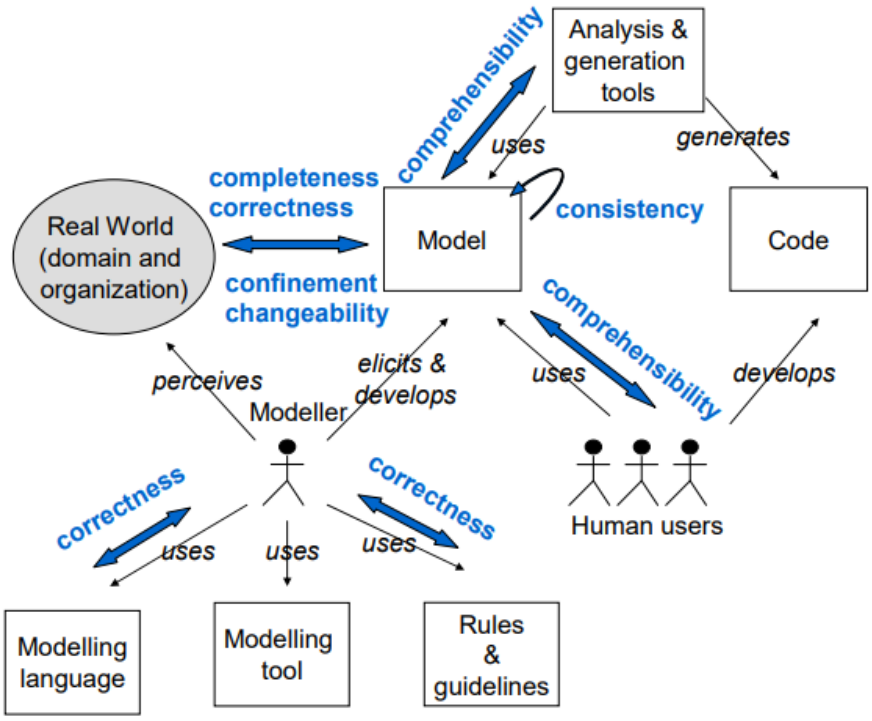


Figure 3.2: Model-based software development with transformation of real world to running software [56]

When it comes to automation, most professionals agree that it should be considered and implemented so that the key resources are freed from repetitive, mundane and low value tasks and instead focus on creative tasks or start to plan the next generation of products. However, when it comes to actual implementation there is very little automation considered or implemented. It is either implemented for its name sake or paid lip service so that it can be claimed that the process is automated but very little gain can be noticed from automation.

The benefits of automation include [57]:

- Repeatability.
  Once a script is coded and tested, it can be executed multiple times with minimal human intervention or effort. The same sequence of instructions can be executed in the same order without the risk of errors.

- Reliability.
  Automation reduces the chance of human errors over time.

- Efficiency.

  After the automation is set up, tested and approved, the same steps can be executed at a much faster rate as compared to the same scripts being executed manually.

- Testing.

  Automation testing can accelerate the amount of test cases executed and the same test cases can be executed multiple times with different values in a shorter amount of time as compared to a manual approach.

- Versioning.

  Automation scripts can be placed on a version control software or repository. In that way, it begins to track all the scripts used and executed for a version release. With manual execution, the procedure documents can be versioned, but the result would still depend on the manual execution.

After the team or project manager decides that automation is the way forward, the next step is to decide which aspect of the project should be automated. This depends on the nature of the project itself, the ability and willingness of the team to automate certain aspects and the potential gain that automation could bring. The typical aspects of a project, suitable for automation, are the following [57]:

- Build and deployment of the system under design.
- Unit test execution and report generation.
- Code coverage report generation.
- Functional test execution and report generation.
- Load test execution and report generation.
- Code quality metrics report generation.
- Coding conventions report generation.

In order for automation to have the maximum impact, it is advisable to automate tasks which have a high frequency (repetitive) and where automation can guarantee reliable results over time. This

is more easily said that done but if automation can be applied to such tasks, the gain would be much more substantial as compared to other tasks. Tasks that occur very frequently during the delivery of a product or software should also be considered for automation. If a task is repeated over time, then every time that task is automated, there is a small gain in human effort which adds up and eventually results in much bigger savings in time and effort. However, the danger is to apply automation to every phase of the SDLC and avoid automation when the cost does not compensate for the benefits. The second pitfall to avoid is to prevent the project from becoming the "guinea pig" for automation. The project manager should remain focused on the deliverables of the project and prevent the project from becoming the test subject for all sorts of automation request. The organization should provide adequate supporting infrastructure for automation or the project may find itself becoming the trial candidate for automation and lose sight of the real objective of the project. Apart from these potential dangers, there are other obstacles which may arise from the need to automation and they are listed below [58]:

- **Stakeholder pressure:**

  There is often pressure from management or other stakeholders who are directly or indirectly investing in the automation. The pressure accentuates on the fact that automation should deliver fast results and reduce costs drastically. These pressures may not give automation a fair chance as results are expected too fast and too soon.

- **Focus on core project requirements:**

  When a project starts, the team and the client want to see progress towards the main deliverables of the project. When time and energy is spent on automating certain aspects of the project, it can be viewed as an unwise allocation of resources, especially before key features are up and running.

- **Lack of management and organizational support.**

  The cost, benefit and how automation is to be rolled out may be lost or a foreign concept to the management team. The whole company may also lack the infrastructure (standards, tools, expertise etc..) to support development work.

- **Lack of follow through.**

The team's morale and commitment may not last to support automation, especially if the results are not tangible initially and take time to materialize.

3.2.1 Automation and code generation

Automated software engineering implies applying computation to software engineering activities. The goal is to partially or fully automate certain activities in order to increase both quality and productivity. This encompasses the study of techniques for constructing, understanding, adapting and modelling both software artefacts and processes. Therefore, automation can cover a large area of the software development life cycle and software engineering. In recent years, it has become more common to develop software by adapting and/or combining existing re-usable software architectures, components such as COTS, frameworks such as Struts and Turbine, and software packages such as ERP and CRM [59]. Even for more established software programming languages like Java and .NET, frameworks such as the Spring MVC or .NET MVC application are favoured [60][61]. Whenever applications have to be coded from scratch, design patterns are also favoured so that the programmer encounters less technical issues and has a faster way to troubleshoot. Design patterns offer known solutions to known and commonly encountered design issues and users can also benefit from more support from online communities. These practices can be regarded as automation or at least partial automation as it reduces human effort and accelerates the development process while also avoiding known pitfalls (and errors).

Automation can take various shape and have varying degree of reducing processing for the user. One example is Quality Analyzer for Requirements Specifications (QuARS), which has been developed by Lami and Trentanni [63], in order to automatically analyse natural language requirements. The tool performs lexical and syntactical parsing of the requirements and the resulting functionalities include defective sentence identification and requirement clustering. In order to detect defective sentences, the tool uses a comparative quality model to provide an evaluation of the requirements that is quantitative, corrective and reproducible. QuARS also uses a series of libraries which evaluate the expressiveness quality of the requirements by assessing the non-ambiguity, the understandability and the specification completion of the requirements. The tool also allows the user to view clusters of requirements by showing a subset of the requirements which group requirements for a similar matter. This feature allows the user to detect

inconsistencies and incompleteness in the requirements. Figure 3.3 shows the inner workings of QuARS:

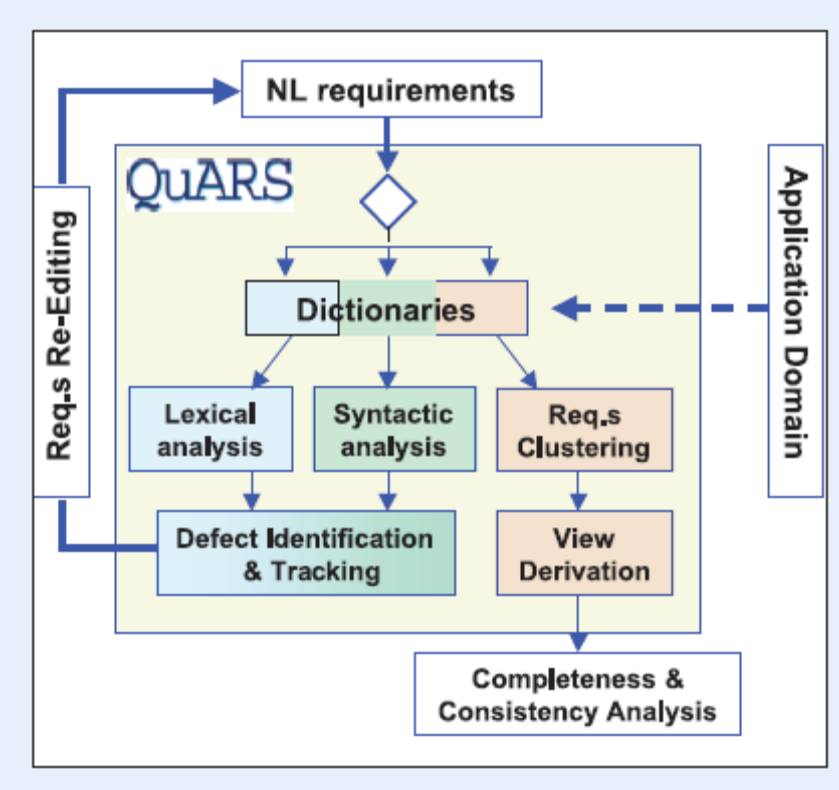


Figure 3.3: The natural language requirements analysis process with QuARS [63]

Having mentioned that automation can occur in various shapes and forms and discussed an example (QuARS) which shows how automation can be introduced in the requirements phase, the next sections analyse automation attempts in specific software engineering tasks. This section focuses on code generation, which allows the developer to produce a portion of the codes automatically without having to write every single line of codes manually. The aim is to produce adequate software code that is comparable to code produced manually by skilled software developers, by using code generation engines. Automated development may lead to a greater use of standardized components, thus increasing software reliability and decreasing the future maintenance costs of software. Automation is also intended to reduce the number of mundane tasks, involved in creating software programs, so that software developers are required to perform more value-added tasks, thus freeing them to focus on tasks that require more creativity [64][65]. It is to be noted that some have questioned the extent to which automation can help software engineers to address the fundamental issues in software development such as complexity,

reliability, and productivity [62]. Automation has, therefore, been both promoted and hailed as well as criticized and occasionally dismissed as a fantasy. Given the real needs of the software generated to work with all the components, it can be understood how the reliability of the generated code is doubted.

The better-known tool for automating software codes from a design are the code generators. Code generators exist in various forms and the underlying premise of a code generator is that given a design, part of the codes can be generated for the developer. For example, given a class diagram, the setters, getters and method names can be generated for the developer i.e. code can be generated from a UML design [66]. The developer can then code the internal logic of the key methods and was saved the trouble of creating all the classes, setters and getters from scratch. Creating a series of classes in programming languages such as Java or C-Sharp (.Net) can be time consuming, mundane and subject to syntax errors which lead to compilation errors. The code generation facility, which is, for example, available in the Visual Studio can help to create the skeleton of the class in a few seconds leaving the developer the time to create the core logic of important methods of that class.

Automation attempts have also affected other aspects of software engineering and has also been used in the later phases of the SDLC [67]. The following sections outline certain parts of the SDLC which have been subject to automation.

3.2.2 Automation in configuration management

Configuration management and version control have been revitalized by new tools like git, software forges and collaboration platforms which makes the software available as Software as a Service (SaaS). Developers and external users find these kinds of repositories useful to view, organize and customize the shared codes. Software forges have rendered version management easier to install and operate and it is no longer seen as a burden but as a value-added service that can be used regularly to manage software, conduct discussions and share information amongst the project team.

3.2.3 Automation in testing and quality assurance

One of the main phases of the SDLC is the quality assurance or testing phase. This activity is the planned activity to detect defects, ensure that the software is compliant with the requirements and

test if the software behaves as expected under various conditions. It is a very important activity as it is the default planned activity scheduled to detect defects and ensure that the software would deliver as per the requirements. Defects and bugs can be found in other phases of the SDLC and reported in the bug tracking tool, but the testing or quality assurance phase remains the primary scheduled activity to unearth these defects. There are three main categories for testing and they are:

- Static and dynamic testing
- The box methods (white box, black box or grey box testing)
- Manual and automation testing.

Static testing is done without executing the program while dynamic testing requires the execution of the program. Static testing is done before the execution of the program while dynamic testing is done after the compilation of the program. Static testing tries to prevent defects while dynamic testing deals with finding and fixing the defects. Static testing requires comparatively a lot more meetings than dynamic testing. Static testing is a verification process while dynamic testing is a validation process. Static testing involves a checklist and process that would be followed while dynamic testing involves the execution of test cases.

Black box testing is also known as closed box testing, data driven testing or functional testing. White box testing is also known as clear box testing, structural testing or code-based testing. In black box testing, the internal working of the application need not be known by the tester whereas in white box testing the internal working of the application is known. In black box testing the testing is based on the external expectations as the internal behaviour of the application is unknown. Conversely in white box testing the internal workings of the application are fully known and the tester can design the test data accordingly. White box testing is the most exhaustive and time consuming of the two. Black box testing is performed by trial and error whereas for white box testing, data domains and internal boundaries can be tested. There is a type of testing known as grey box testing which is a mixture of the black box testing and white box testing.

There is then manual testing where the user executes the test cases manually, without the help of any tools. In automation testing the user writes a script and uses another software or tool to test

the piece of software that needs to be tested. The main differences between manual testing and automated testing are listed below:

- Automation is faster than manual testing, which is considered to be time consuming and tedious.
- Automation testing may need less testers as there are fewer manual tasks, but automation testing requires skilled testers to write the script and get the process started.
- In automation testing, the programmable script can be used to fetch hidden information and execute more complex scenarios in one go.
- If there are no errors in the script, automation testing is considered to be more reliable, as test cases can be executed over and over with minimal risk of failing to replicate the same steps.
- Both automation testing and manual testing can be used for Unit testing, Integration testing, System testing and Load testing.

The focus of this chapter is automation and therefore a greater emphasis would be laid on automation testing than other aspects of quality assurance and testing. Software testing automation involves the automation of software testing activities including the development and execution of test scripts, verification of requirements being tested and the use of automated testing tools to do so [68]. Manual testing can be tedious and time consuming and for that reason automation testing is preferred. Automation testing can increase efficiency for repetitive steps to reproduce system functionality, especially in regression testing, where test cases are executed iteratively and incrementally after changes made in the software [69]. Testers must carefully define and analyse the testing automation strategy in a software project as the success of automation requires the knowledge of testing tools and verification of tests results to ensure that the cost of automation is more cost effective than manual testing [70].

In recent years agile methodologies have grown in popularity and therefore automation testing in the context agile methodologies should be considered. Automation testing, in agile methodologies, can be expensive to implement at first and usually the fruits of the investment can be borne in the long term. The agile methodologies are preferred as they allow the software engineers to rapidly

accommodate changes in the requirements and prioritize the development of features through executable code rather than extensive written documentation. Given the iterative process of the agile methodology, collaboration with the customer is favoured instead of following rigid plans and contract negotiations [71]. Automated testing tools need to be used efficiently in order to provide quick responses to the requirement changes as well as effective interaction with the development team [72].

Automation testing in agile is not a straight forward task and according to Crispin and Gregory [73], the following issues can impact the success of testing automation:

- Programmers tend to overlook testing as there is a test team to catch bugs.
- The “Hump of Pain”, i.e the learning curve for testers to learn about tools and code.
- Initial investment (acquiring tools and training people) can be prohibitive.
- Code and interface can be constantly changing, making the GUI automation unfeasible.
- Legacy systems codes are not designed for testability.
- Testers without programming background and programmers without previous experience in testing, can be apprehensive of the process.
- By sticking to old habits, manual regression tests can be favoured instead automated regression tests suite.

There can be challenges and apprehensions for teams to adopt automation testing. Once they have done so, the gains can be substantial as regression testing or testing of certain modules can be much faster. In recent years, there has been a growing adoption of web-based applications and logically these applications have also been affected by automation testing. Whether the architecture is closer to service-oriented architectures or micro-services or more classic distributed systems, the trend is to use web-based application. Regardless of the technology stack, like Java web applications, .Net suites of products or UI technologies (like JavaScript variants, HTML5 or CSS3), the tendency is to use web based distributed architectures. Therefore, the next paragraph analyses how automation testing is used for testing web-based applications.

Web testing or web automation testing are the names given to the activities of testing web-based applications either manually or with the use of automation testing tools or a combination of both. The best-known tool set is the Selenium IDE suite of products. There are also plenty of other which can be used for web-based automation testing and the table 3.1 gives a description of all of them, including Selenium.

Tool Name	Description
Selenium	A suite of products comprising of four tools: Selenium IDE, Selenium RC, Selenium Webdrive and Selenium grid. It is a portable and open-source automated testing suite which has been improved since its release in 2004 and is now a strong tool for web-based automation testing.
HP–QTP	HP–QTP is part of Hewlett Packard (HP) quality centre tool suite which provides regression and functional testing automation for major software environment and applications.
FitNesse	Fitnesse is an automation testing tool used to write, organize and execute table-based tests. Fitnesse is wiki wrapper over the fit sever that provides a web interface to the test suite. It also allows customers, testers, and programmers to discover what their software should do and automatically compares that to what it actually does. It is also compatible with Junit.
Watir	Watir is an acronym for Web Application Testing In Ruby. It is an open source family that uses ruby libraries for automation web browsers and allows testers to write tests that are easy to read and maintain.
TestComplete	TestComplete is an automated testing tool which allows testers to create, manage and run tests for any Windows, Web or Rich Client software.
LoadRunner	HP LoadRunner is an industry standard based an automated performance and test automation product from HP for load testing of application, which examines the system performance and behaviour. HP LoadRunner works by using the virtual users and simulates thousands of concurrent users to put the application

	through various real-life user loads and analyses the results to discover the particular behaviour.
TestNG	TestNG, which refers to "Testing, the Next Generation", is a testing framework that is inspired from JUnit6 and NUnit7.
TOSCA	TOSCA is a test suite for the automated execution of functional and regression software testing. TOSCA test suite includes integrated test management, design, execution and data generation toolset for functional and regression tests.
SilkTest	SilkTest is an automation tool that is specifically designed for regression and function testing. SilkTest provides a flexible and robust test scripting language that is built in recovery system for unattended testing, and SilkTest can be used across multiple platforms, browsers and technologies. Silk Test offers test planning, validation, management, and direct database access.
WinRunner	The WinRunner software (from HP) is an automation Functional GUI testing tool. This tool allows the user to capture, verify and replay UI interaction as a test script.

Table 3.1: Automation test engines

3.2.4 Automation in software building

Automation can be applied in all the activities and processes related to building, deploying, and operating of software systems. A programmer can find out the software release from a version management system, download all needed external components and libraries and prepare for the deployment of the final product by running a Maven script. Some of these facilities are also coupled with Integrated Development Environments (IDE's) like Eclipse and achieve the same results through an intuitive Graphical User Interface (GUI). Building on top of that some framework like Jenkins [74] integrates some of the previously mentioned tools and offer an environment which can be configured to generate the product ready to be deployed, whenever a new version of a single source code file is pushed to the version management system. This is known as continuous integration [75]. Automation can also be applied to the operation phase for the management of applications. The introduction of virtualization and remote-control

mechanisms and the availability of public and private clouds have made this possible. For example, Puppet [76] is an example of a tool supporting application management, while Nagios [77] can be used for monitoring the application state.

3.2.5 Automation as DevOps.

The increasing level of automation enables software developers to fulfil the requirements of business-critical applications in terms of availability, reliability and response time. It also creates a bridge between the development and the operation activities which is often known as DevOps [78]. DevOps is driving important organizational change within the software industry and as reported in [79], organizations like Amazon are choosing a "per service" organization, where cross-functional teams are responsible for both the development and the operation of a set of services. The performance metrics for such teams are focused on the quality and stability of the final services rather than the productivity of individuals or other traditional productivity metrics.

So far, the various aspects of the SDLC which have been subject to automation have been listed and briefly explained. It is noticed that the main areas of automation focus mostly on the later stages of the SDLC and include configuration management, quality assurance, software building and DevOps. However, as mentioned in the previous chapter, defects can occur at any stage of the SDLC and the defects arising in the earlier stages of the SDLC are most costly to fix and more detrimental to the project. Automation and its benefits can also be applied to the whole of the SDLC, i.e in the earlier stages of the SDLC too. The aim would then be to produce software with less defects and high-quality factors. Thus, the next sections provide a review of some of the automation strategies for capturing the requirements correctly and, in some cases, attempt to produce some design elements.

3.3 Automation in the requirement and design

As explained in the previous chapter, the cost of software malfunction is the highest in the earlier stages of the SDLC. Therefore, it is more worthwhile considering and applying automation techniques to that portion of the SDLC. The following sections provide a walkthrough of the various techniques of automation and tools that have been crafted by other teams in order to automate the translation of requirement into design artefacts, with the aim of reducing defect

injection or ensure that the requirements are properly managed so that they can be translated effectively into design artefacts.

3.3.1 Defect management techniques in requirement engineering (classical approach)

IBM proposes the Rational suites of product to its clients to manage the requirements of an IT project. Requirements and test cases can be documented, tracked and linked using Rational DOORS or Rational DOORS Next Generation [80]. The Rational suites of products mostly help the client to track the requirements, link them to test cases, allow multiple users to modify and update the requirements and create views to visualise different versions of the requirements. Similarly, Accenture proposes the Accenture Requirements Engineering Suites (AcRES) of products to its customers for them to manage their requirements [81].

Other researchers [82][83] advise on a series of methodologies which can help to reduce requirement defects, such as:

- Joint Application Development (JAD) originally designed to bring system developers and users of varying backgrounds and opinions together in a productive and creative environment. JAD recommends having meetings between the software company and the client to refine the requirements [84].
- CASE (Computer Aided Software Engineering) and requirement engineering tools, which are engineered to assist throughout the development life cycle and improve the quality of the software [85].
- Quality Function Deployment (QFD) which involves listening to the customer exigencies (Voice of Customer) and translating those in design targets and quality assurance targets for the project [86].

3.3.2 Need for textual analysis

The above-mentioned commercial tools available from IBM or Accenture are requirement management tools which allow the clients to centralise and manage the requirements. However, these tools do not assist in the requirement elicitation process and it is still the responsibility of the business analyst to source the requirements. There is thus no guarantee that requirements defects

will be reduced as there is no validation on the quality of the requirements inputted into the system. The other methodologies mentioned can assist in the requirement elicitation process and can help the business analyst to ensure that key requirements or even design features and quality assurance targets are captured. However, none of the methodologies (like JAD, CASE tools or QFD) deal with the descriptive nature of the requirements and hence the requirements may still be inconsistent or ambiguous.

For these reasons, it is necessary to look at other techniques which could analyse the texts captured by the business analyst with the aim to detect any inconsistencies and ambiguities. It is also worthwhile investigating techniques which could automate the interpretation of natural language requirements to conceptualise the design. The next section analyses a series of approaches which make use of textual analysis and Natural Language Processing (NLP) to parse the requirements to make sure they are consistent and unambiguous. Some of these approaches have also attempted to automate (or partially automate) the design process by modelling the requirements with UML design artefacts.

3.4 Software requirement: Textual analysis techniques

3.4.1 Overview

There are a number of defects management techniques which textually analyse the requirements with the aim to assist in the design process [87-90]. They employ Natural Language Processing (NLP) techniques to parse the requirements and generate models, which can be used in the design process. These models are typically UML artefacts, which can be used as the blueprints for software construction.

While NLP has been applied to various software engineering fields (such as machine translation, natural language text processing and summarization, user interfaces, multilingual and cross-language information retrieval (CLIR), speech recognition, artificial intelligence, and expert systems) [91], IT practitioners were unsure if NLP could be applied to requirement engineering. Ryan [92] claimed that NLP would not be able to understand the requirements. Furthermore, even if NLP could be used to understand the requirements, NLP would not be able to infer the requirements which are not explicitly written down but are understood as part of "common domain

knowledge". Therefore, NLP would not be applicable to requirement engineering, especially for large systems.

However, further research work has continued, and a number of investigations on the applicability of NLP for requirement engineering have appeared in literature [89][93][94]. These include AbstFinder [95], which is a prototype tool that can be used to find abstractions in natural language text to be used in requirements elicitation. Amongst all the published work, the one accomplished by Kof [96] can be regarded as the first to formally dismiss Ryan's claims. Kof demonstrated that NLP was mature enough to be applied for Requirement Engineering as the aim of NLP is not to understand the requirements, but to extract concepts from the requirements. In the approach proposed by Kof, the system engineer is solicited to validate the parses at various stages and can thus ensure that the requirement documents, containing all the key requirements vital for the software, are written down. The method proposed by Kof uses a series of existing techniques in order to formulate an approach which can be used to parse natural language requirements so as to ensure that the requirements are correctly described, as unambiguously as possible. Figure 3.4 summarises the integrated approach used by Kof.

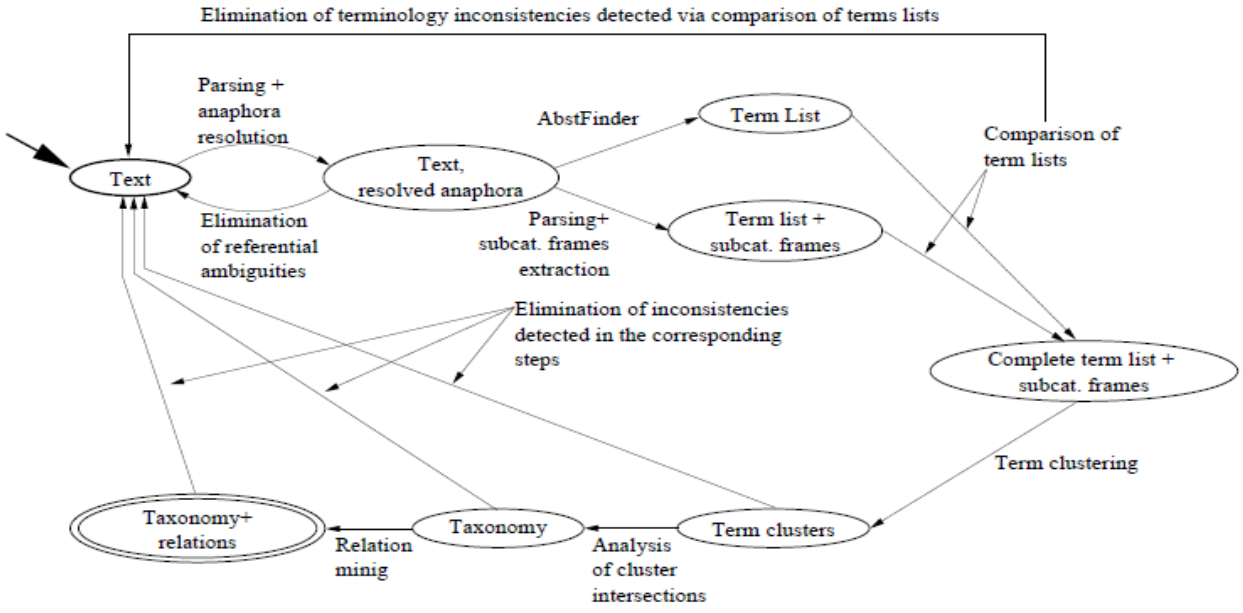


Figure 3.4: Integrated ontology extraction approach [96]

The approach can be summarized as follows:

- Parsing and anaphora resolution.

The cross-sentence references are removed, mostly by replacing the pronouns by the actual words that they are referencing. The domain expert needs to validate the terms replaced to ensure that there are no errors.

- AbstFinder

  AbstFinder [95] considers the sentences just as character sequences and is used to extract character sequences that are common for at least two sentences.

- Parsing and sub categorization frames extraction

  After the first two steps, the text needs to be parsed again, or at least the sentences which have been altered by the first two steps (Anaphora Resolution). In order to achieve this, the parser by M. Collins was used [97]. Figure 3.5 shows an example of how the parser extracts the terms, which is summarized below.

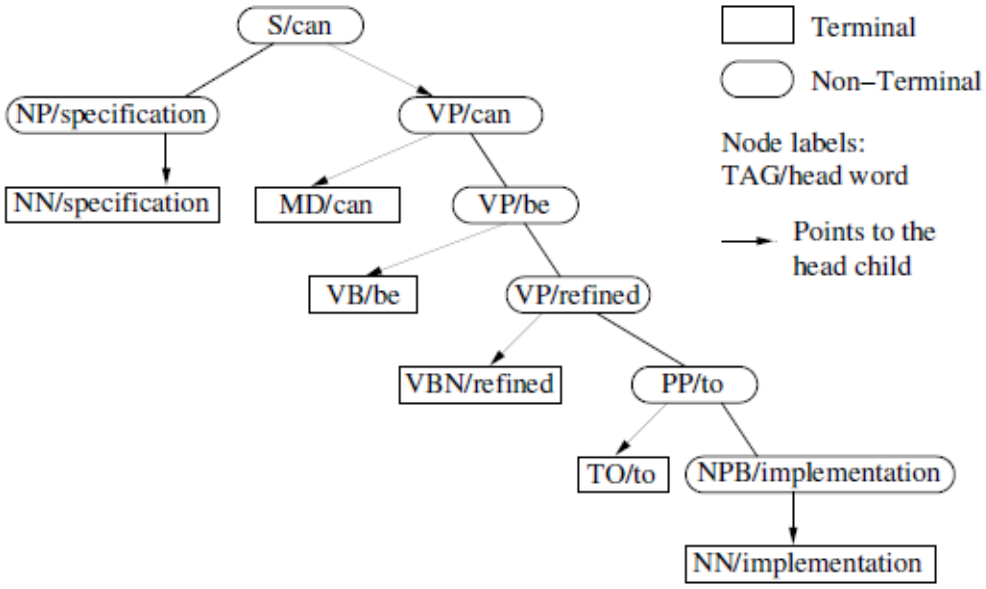


Figure 3.5: Parse tree for the "Specification can be refined to implementation" [96]

- Comparison of term lists

  The AbstFinder step will retrieve terms which occur at least twice in the text and sub categorization frames extraction can extract terms that occur in grammatically correct sentences only. By comparing the two lists, the user can narrow the terms which have

occurred at least twice in the text, but solely in grammatically incorrect sentences. The user therefore needs to review the text to check if there are terms which have been omitted.

- Term clustering

  In order to cluster the terms, it is important to find the related or similar terms. The three types of similarity exploited are: contextual, lexical and syntactic term similarity. There are weights assigned to each similarity, with relations to the analysed texts. The user will have to review the clustered terms to ensure that the terms are clustered accordingly.

- Taxonomy building

  Concept clusters constructed previously are used for taxonomy construction by joining intersecting clusters to larger ones. The resulting larger clusters represent more general concepts.

- Relation mining

  There is a potential association between two concepts if they occur in the same sentence. Each potential association again has to be validated by the requirements engineer.

The integrated approach aims to establish an application domain ontology which is validated by the user as it is being constructed, during the requirement engineering phase. It makes use of a series of techniques and the weaknesses of the individual techniques are compensated by the strengths of the other techniques used. The resulting document is a validated application domain ontology and a corrected textual specification, free from terminology inconsistencies.

The work accomplished by Kof demonstrates that natural language processing (NLP) is mature enough to be applied to requirement engineering and also demonstrates that a series of existing techniques can be applied in an integrated approach to parse requirements, written in natural languages. NLP can hence be viewed as being mature enough to assist in the requirement engineering phase, with the help of manual intervention required. However, the approach does not assist in the design process and there are other approaches which automate or partially automate the design process. A number of these frameworks are discussed in the following section.

3.4.2 Frameworks to model requirements using NLP techniques

3.4.2.1The RACE framework.

Mohd Ibrahim and Rodina Ahmad propose the RACE framework [87] which uses NLP techniques to parse the requirements and also defines a set of rules which are then used to start deriving class diagrams from those requirements. The RACE framework is automating part of the design process. The architecture of the RACE framework is detailed in Figure 3.6.

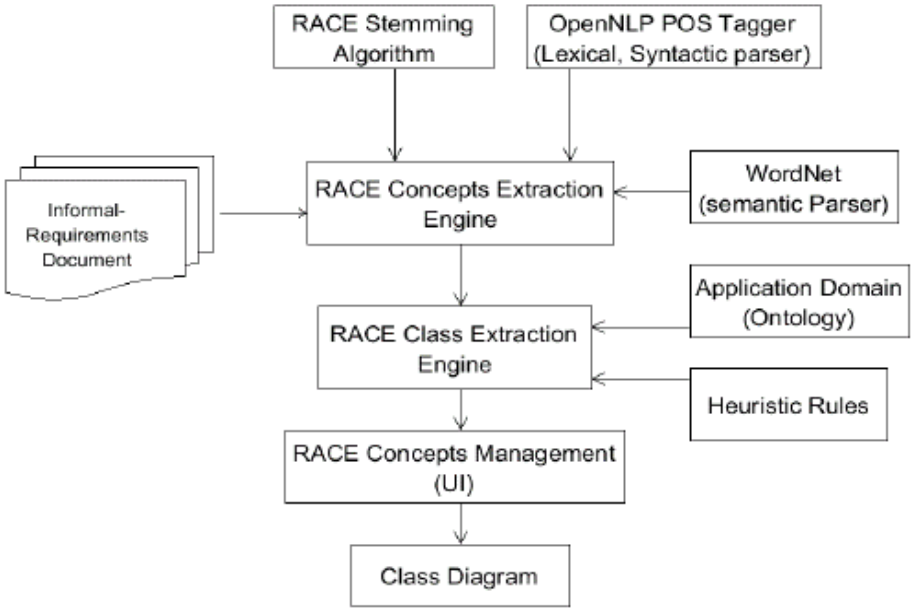


Figure 3.6: RACE system architecture [87]

The RACE framework makes use of the six main components, some of which are internal to the system while others are third party systems:

- OpenNLP Parser [98], which is an open-source parser, provides the syntactic and lexical parsers of the RACE framework.
- RACE Stemming Algorithm which mostly returns words in their simplest form by removing affixes and suffixes. For example, the word “strategies” would be returned as “strategy” and the word “books” would be returned as the word “book”.
- WordNet [99] is used to evaluate the semantic correctness of sentences generated. It is also used to extract synonyms which are semantically related to each other.
- The Concepts Extraction Engine makes use of the first 3 components enumerated above and then validates the sentences through 9 rules to derive the key concepts.

- The Domain Ontology is used to improve the performance concepts identification.
- The Class extraction Engine is the most complex engine which makes use of eight rules to identify classes. It also makes use of two rules to identify the attributes and six rules used to identify the relationship between the classes.
- RACE Concept Management (UI). This engine mostly deals with the UI of the system which the user interacts with. The user can create, print, save and analyse requirements through this engine. The user can also add, delete and rename classes and relationships of class diagrams. Most of the user interaction as well as the graphical user interface is handled by this component.

The RACE framework uses a blend of NLP techniques and domain ontology techniques to extract class diagrams from requirements written in natural languages (English and Malay). The user is presented with a class diagram which he can modify, view and organize concepts and relationships through the RACE user interface.

3.4.2.2 The Circe framework

Vincenzo Ambriola and Vincenzo Gervasi propose the Circe framework [89], which uses NLP techniques and a series of validation techniques to assist the translation of requirements into (semi) formal models. The architecture of Circe is detailed in Figure 3.7.

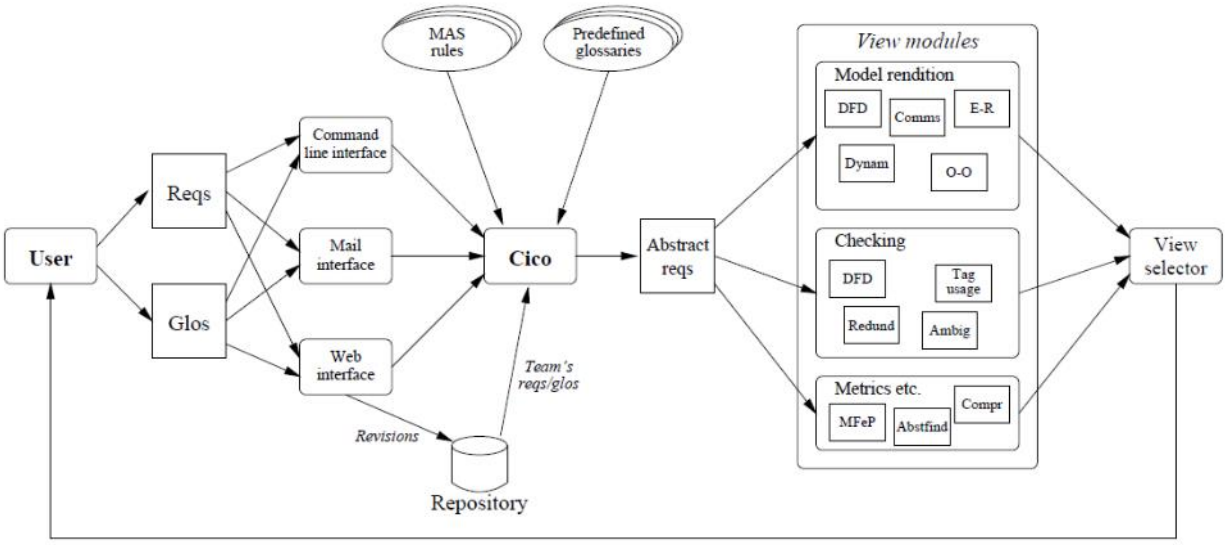


Figure 3.7: Architecture of the Circe environment [89]

The user can use the system through a command line interface, a mail interface or a web interface. The system will accept a set of requirements as input, but should also be provided with a glossary of terms. The glossary of terms is defined following some syntactic conventions and can be a set of words describing and classifying all the domain specific and system specific terms of the requirements.

The core of the solution which actually parses the requirements and transforms them into a series of (semi) formal models is the Cico engine. The Cico engine uses a custom algorithm to parse the Natural Language, but a series of other parameters and functions, listed below:

- MAS (Model Action Substitution) Rules

  The MAS rule(s) can be summarized as such: $\text{MAS-rule } r = \langle m, a, s \rangle$ applied to a requirement *t* will yield the following: if a fragment of *t* matches *m,* action *a* is executed and the matching fragment of *t* is replaced by *s*. The model matching algorithm is a fuzzy one and it associates to every possible matching a score of similarity with the model. The recognition power of the tool is hence enhanced and it also makes use of a weightage system to determine the mapping.

- Predefined glossaries

  The predefined glossaries include a number of relations taken from common sense, generally expressed as verbs (“to send”, “to receive”, “to compute”…) whose semantics and correspondence with the domain’s underlying model is assumed to be intuitive. Thus, they can be used to define terms for different languages.

- A repository

  The repository will accept glossary terms and requirements and will feed the data to Cico when solicited.

The output of the Cico engine is hence a series of abstract requirements, which are then fed to View module(s) which will produce a graphical representation of that data, which are mostly semi-formal models. The views usually use the abstract requirements but can also access the source requirements, the glossaries or the output from other modules (like the repository) to produce its output.

There are essentially three types of views:

- The Model rendition views create the graphical representations of the (semi) formal models such as Data Flow Diagrams (DFD's), Communication diagrams, Entity Relationship (ER) diagrams, Dependence diagrams and Object-Oriented paraphrase diagram.
- The Reporting (Checking) views mainly perform checks and validations on the outputs from the model rendition views.
- Metric and other views

There are three metric views which can help to evaluate the accuracy of the system. The Circe framework integrates natural language processing, modelling and validation aspects in an aim to assist in requirement elicitation and validation.

3.4.2.3 The UMGAR Framework

Deeptimahanti et al. [88] propose the UML Model Generator from analysis of Requirements (UMGAR) framework which aims to provide semi-automated support for developing both static and dynamic UML models from Natural Language requirements. The architecture of the UMGAR framework is shown in Figure 3.8.

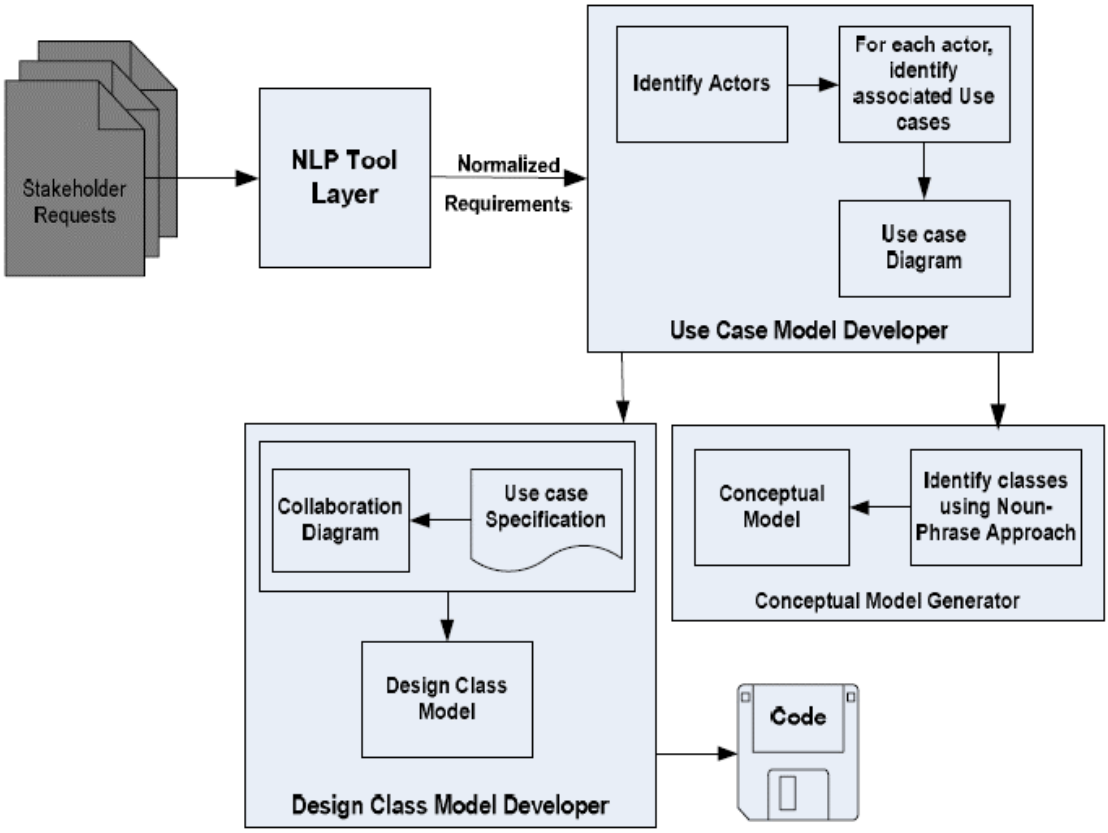


Figure 3.8: Process architecture of UMGAR [88]

There are two main components in the UMGAR framework and they are:

1. Normalizing requirements component (NLP Tool Layer)
2. Model Generator Component

The Normalizing requirements component (NLP Tool Layer) is made up of two sub-components:

1. The Syntactic Reconstruction component.

   Based on their previous work [100][101][102], Deeptimahanti and Sanyal propose 8 reconstruction rules to transform each Natural Language (NL) sentence so that they are in the form of "Subject: Predicate" or "Subject: Predicate:Object".

2. The other component makes use of the following tools on the normalized requirements:
   a. Stanford Parser [103] is a Natural Language (NL) parser which is used to extract artefacts like actors, use-cases, classes, methods, associations, and attributes.

   b. WordNet [99] is used to perform morphological analysis for converting plurals into singulars. They have noticed that WordNet2.1 does not work well when addressing the plural of compound words such as "Verification officers" and they have coded an algorithm using 21 rules to ensure that compound words were not being incorrectly altered by this step.

   c. JavaRAP [104] which is used to resolve pronouns up to third person pronouns.

The Model Generator component is made up of three sub-components:

1. Use case model developer. This part of the framework identifies subject and object as actors, predicates as use-cases, associates as actors and use-cases from the normalized requirements.

2. Analysis Class Model Developer. This component identifies all candidate classes from requirements and generates analysis class model by attaching attributes and methods with associated class object. It also eliminates redundant classes by checking a glossary and eliminates ambiguity by carrying out a morphological analysis using WordNet2.1.

3. Design Class model developer. There is a series of eight rules which are applied so that collaboration diagrams are generated. Those collaboration diagrams are then used with the help of the Stanford parser to generate the design class model.

The UMGAR framework is then tested on a case study and the framework is able to generate a use case model, an analysis class diagram and a design class model. These UML diagrams are generated as an XML Metadata Interchange (XMI) [105] file and therefore can be imported by any UML tool which has an XMI Import feature. The UMGAR framework still requires human intervention for the elimination of irrelevant classes and identification of aggregation/composition relationship among objects, and is as such presented as a semi-automated tool to assist the translation of requirements and generate UML based analysis and design models.

3.4.2.4 Other researches

Other researches include the work carried out by Zhou et al. [90] who also proposed a framework which uses NLP techniques and domain ontology building techniques to extract class diagrams from natural language requirements. It is based on the fact that the core classes are always semantically connected to each other by one to one, one to many, or many to many relationships in the domain. It finds candidate classes using NLP techniques and domain ontology is then used to refine the results.

A Natural Language-Object Oriented Production System (NLOOPS) LOLITA was proposed by Mich [106] and it generates Object Oriented analysis models. These models are obtained by parsing natural language SRS documents and from SemNet. Nouns are considered as objects and relationship among objects are identified using links. This approach lacks accuracy in selecting objects for large systems and cannot differentiate objects and attributes.

A tool for constructing an object model automatically based on pre-specified key words in use-case description is proposed by Börstler [107]. The nouns are transformed to objects and the verbs in the key words to behaviours. The approach requires excessive user interaction to correctly match a behaviour to an object. A tool using syntactic knowledge by extracting objects, methods and associations to generate object diagram from natural language requirement document is developed by Nanduri and Rugaber [108]. However, the resulting models have to be validated manually and user needs to possess vast domain knowledge.

CM-Builder analyses requirements text and builds a Semantic Network in order to create an initial UML Class Model [109]. This model can then be visualized using a graphical CASE tool by converting it into standard data interchange format, CDIF. The user can make further adjustment

so that the final UML models generated match the user's expectations. Yet, it is to be noted that CM-Builder makes an extensive use of NLP techniques to simply generate analysis class models.

Linguistic assistant for Domain Analysis (LIDA) [110] identifies model elements through text analysis and these models are validated by refining the text descriptions of the developing model. When using LIDA, continuous user interaction is needed to produce the models as it identifies only a list of candidate nouns, verbs and adjectives. These then need to be categorized into classes, attributes or operations based on user's domain knowledge. Popescu et.al [111] have developed a tool with the aim of identifying ambiguity, inconsistency and under specification in requirement documents. This is achieved by creating object-oriented models automatically by parsing natural language software requirements specifications according to a constraining grammar. The reviewer is then tasked to detect ambiguities and inconsistencies from the diagrammed UML models.

There are relatively few attempts at providing tools for generating behavioural models like sequence or collaboration models from NL use-case specifications, from which design class model can be generated. Li [114] proposes a semi-automatic approach to translate narrative use-case descriptions to sequence diagrams using syntactic rules and parser. Eight syntactic rules are proposed to handle simple sentences, and the user is then solicited for sentences which have insufficient data to handle different types of verb phrases.

Use Case Driven Development Assistant Tool (UCDA) [113] generates Class Model by analysing natural language requirements. It assists in creating use-case diagrams, use-case specifications, robustness diagrams, collaboration diagrams and class diagrams. The main drawback of UCDA is that it depends on Rational Rose which is a very expensive environment, to visualize UML models. Montes et.al [115] and Diaz et.al [116] developed a tool to generate conceptual model, sequence diagrams, and state diagrams by analysing a system's textual descriptions of the use-case scenarios in Spanish language.

Yue et.al [117] proposed a method to generate activity diagrams from use-case specifications using transformation rules. There is also a commercial tool named Ravenflow [118] which provides a mechanism to generate activity diagrams (process diagrams) from structured text written using rewriting rules. The main disadvantage of this tool is the limitation in representing alternative flows.

3.5 Analysis of existing frameworks

Natural language requirements generally describe four main categories of requirements which pertain to data (or describing flow of data), the process flow, the communication and non-functional requirements. Non-functional requirements are not modelled but the three other categories of requirements are. In order to define all the features and characteristics of a piece of software it is important to model the possible behaviours and interactions of the system, especially for a distributed system. Usually, data requirements are modelled by an Entity Relationship Diagram (ERD) or a class diagram. The requirements describing a process flow are usually modelled by a Data Flow Diagram or illustrated with a flowchart. The communication sequences within a system are usually modelled using a Sequence diagram and the non-functional requirements are not modelled.

The following table summarises how the existing approaches cater for the above-mentioned categories of requirements:

Feature	Kof' approach	RACE	Circe	UMGAR	Zhou et al	LOLITA	CM Builder	LIDA	Popescu et.al	UCDA
Normalise Requirements	N	N	N	Y	N	N	N	N	Y	N
Data models	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
Process models	N	N	Y	N	N	N	N	N	N	N
Communication models	N	N	Y	N	N	N	N	N	N	N
XMI Support	N	N	N	Y	N	N	N	N	N	N
Evolving system (learning system)	N	N	N	N	N	N	N	N	N	N
Abstraction of the design	N	N	N	N	N	N	N	N	N	N

Table 3.2: Comparative analysis of the existing approaches

Whilst Kof's approach is able to eliminate inconsistencies and reduce ambiguity in natural language requirements, it does not automate the design process. All of the remaining approaches or frameworks go one step further and model the requirements. The RACE framework, UMGAR and the approach proposed by Zhou et al. only produce data models. The Circe framework is able

to model all three types of requirements but the communication model is a communication diagram and not a sequence diagram. Furthermore, the Circe framework was developed in 1997, at a time when service-oriented architecture had not been used.

It is also noticed that UMGAR is the only one which normalises the requirements. Normalisation of the requirements prior to the extraction object-oriented concepts makes the framework more robust and thus more easily applicable to large and complex systems. Popescu et al. also change the requirements by using a constraining grammar. This forces the user to write the requirements in a form that can be more easily interpreted by the framework. UMGAR is also the only framework to output the design artefacts as XML files, compliant with XMI standards. This means that an output from UMGAR can be viewed in other tools which also have an XMI import feature. The user is not restrained to viewing and manipulating the design artefacts through the UMGAR user interface alone. For the other frameworks, the user has to view and manipulate the outputs through the framework's own user interface or in output files.

LOLITA parses the requirements and identifies nouns as objects and the relationship between objects. The approach decreases in accuracy for large systems and fails to differentiate between objects and attributes. Börstler [107]'s approach transforms nouns to objects and verbs to behaviour based on pre-specified key words on use cases, but that approach depends heavily on user interaction. Similarly, Nanduri and Rugaber [108]'s approach requires a lot of user involvement and a vast domain knowledge from the later to produce accurate results. CM Builder creates an initial UML class Model and the user can adjust the model so that it is satisfactory, but it uses extensive NLP techniques to produce analysis class models. LIDA also identifies model elements from text analysis and the models are refined by the user. The models are categorized in classes, attributes or based on domain knowledge. Popescu et.al [111] 's approach creates object-oriented models by automatically parsing software requirements specification according to a constraining grammar, which also depend on the user to validate the results.

The tools generating behavioural models include Li's [114] approach which is a semi-automatic approach to use-case descriptions to sequence diagrams. This is achieved by using eight syntactic rules to handle simple sentences and the user is solicited for sentences with insufficient data. UCDA [113] creates use-case diagrams, use-case specifications, robustness diagrams, collaboration diagrams and class diagrams. UCDA uses Rational Rose which is a very expensive

environment. Montes et.al [115] and Diaz et.al [116] developed a tool to generate conceptual model, sequence diagrams and state diagrams by analysing a textual description of the system specified in the use-case scenarios for the Spanish language. Yue et.al [117] proposed a method to generate activity diagrams from use-cases and the commercial tool named Ravenflow [118] provides a mechanism to generate activity diagrams (process diagrams) from structured text written using rewriting rules, but the tool is the limited in representing alternative flows.

All of the existing attempts are either directly aiming to extract object-oriented concepts or produce UML diagrams. This is not a flaw but none of the existing approaches have been engineered specifically for service oriented distributed systems. Many of the automation attempts reviewed in this chapter are designed for web-based applications as these applications are more and more popular and more widely adopted. Therefore, targeting an architecture which can be abstracted can be more appropriate for distributed web-based applications. The existing approaches do not propose an abstracted design and may not be well adapted for distributed systems. It was also observed that the existing frameworks are stagnant to a certain extent in the sense that the algorithm and parsing capacity of the framework does not evolve over time. They would produce the same results when presented with the same set of requirements. In some case the user could modify the results, but the initial parse would be the same. For example, the user could manually eliminate redundant classes in the UMGAR framework, but the words identified as classes would always be same for a given set of requirements. It is therefore worthwhile exploring a dynamic system which learns over time through user interaction and grows more accurate.

This research aims to produce a framework which will model the requirements by proposing a design for the potential software. Building on that, an abstracted architecture will then also be produced, resulting in a design that would be better suited for a distributed system. It is also proposed to couple the framework with a learning system which will allow the parses to grow more accurate over time. In so doing, the approach will provide a comprehensive tool to model the natural language requirements, better adapted for modelling distributed systems. The outputs from the tool will be three sets of design artefacts: a class diagram, a use case diagram and a sheet showing the abstraction and relations between the layers. The resulting enterprise architecture and the technical architecture are more likely going to be aligned with the requirements, thus reducing the likelihood of requirement and design defects.

3.6 Summary

Aiming to minimize the overhead of mundane, repetitive and low-value tasks in software engineering, the help of automation and partial automation can be solicited. Automation can be helpful in software engineering tasks as it can help to speed up certain tasks, process a larger volume of work, ensure traceability and consistency across several tasks. In this chapter, many types of automation currently in use in the different phases of the SDLC were explained. Automation has been applied to code generation, configuration management, version control, automation testing, DEV OPS and software building. As explained earlier, the cost of software malfunction tends to increase when the defects are detected earlier in the SDLC and therefore it is worthwhile investigating approaches which aim to reduce requirement and design defects. Therefore, automation attempts to translate requirements into a design were also reviewed in this chapter.

The interpretation of requirements and translation of these requirements into a design are an important aspect of the software development lifecycle. The manual approach can be tedious and time consuming, and therefore some kind of automation can accelerate the process. Automation is not an exact science and very few off-the-shelf solutions exist that can directly fulfil the needs of most users. However, there are several approaches, techniques and tools which can be used to accelerate the process of requirement translation and elaboration of the design. This chapter has listed the main approaches, techniques and tools, alongside their main advantages and disadvantages. It was pointed out how they achieve their goals and create design solutions for a given set of requirements. It was also pointed out that design produced are not abstracted and may not be the best solution for distributed systems such as service-oriented architecture or micro-services, which are growing in popularity. The aim of this research is to propose an approach which is better adapted at automating or partially automating the translation of the requirements into a design and producing an abstracted architecture. The approach would also include a learning system which is intended to learn and grow over time so that the system grows more accurate with more data. By abstracting the design, the solution would be better adapted to be used for distributed systems. The next chapter starts to list all of the key components of the proposed solution. The key components are explained, how they interact with each other and how the results are produced.

CHAPTER 4: THE PROPOSED FRAMEWORK

4.1 Introduction

The previous chapters have painted a picture of the software malfunction and how the software malfunction can prove to be costly and how defect leakages can impact software projects negatively. Software is more and more invasive in our daily life and going forward software is bound to play a growing role in the society. As a result, software malfunction is also bound to have a larger impact on the society. The impact may not necessarily be commercial, but can be social, economic, ethical and also legal. The previous chapters have also covered an analysis of the cost of the software malfunction, the classification of defects, and how the impact of defects injected in the earlier phases of the SDLC can be the costliest to fix. The cost of software malfunction can have a direct monetary impact, but can also have an indirect impact such as the loss of business due to loss in faith in the reputation of the company or may also include regulatory fines.

The previous chapter explained how existing approaches, have been developed to facilitate requirement elicitation process and assist the business analyst in making less mistakes. The approaches, methods and methodologies can be commercial tools or practical logical methods which help the business analyst keep track of the requirements and follow up with the business side of the project so as to ensure that the project is still in line with solving the business problem. These tools (Rational DOORS, Rational DOORS Next Generation and Accenture Requirements Engineering Suites (AcRES)) do a great job at assisting the management of the project and ensure timely tracking of the project deliverables. However, the tools do not deal with the inherent nature and ambiguity of the natural languages. Natural languages such as English, French, German, etc. are subject to ambiguity and sometimes to confusion. Various other approaches developed, have used natural language processing (NLP) parsers and custom algorithms to attempt to extract meaningful concepts from the natural language requirements. These concepts are then used by a custom algorithm, coupled with a data dictionary in order to enhance the output of the framework.

Some of the frameworks can then create unified modelling language (UML) diagrams while others do not. The aim is not to understand the requirements but to extract concepts from the text.

The advantages and drawbacks of the existing approaches were reviewed and the need for a new approach was explained. Therefore, the goal of the research work is to produce a framework which can produce a design, with a capacity to learn and grow over time and better adapted for distributed systems. The framework is detailed over the next sections of the chapter, and the different parts of the framework are explained elaborately. The next section (4.2) outlines the main components of the framework and how they interact with each other. Section 4.3 outlines the proposed framework and the NLP (Natural Language Processing) component, section 4.4 describes the design extraction component and section 4.5 describes the learning system which comprises of the ontology and neural network. Section 4.6 describes the user validation process and the chapter is concluded in section 4.7.

4.2 Proposed framework

The analysis of the existing approaches was explained in chapter 3 and the advantages and drawbacks of each one of them were outlined. The approaches and techniques assist in the requirement elicitation process and automate or partially automate the elaboration of a design from these requirements. However, all of these tools have focused on object-oriented programming and did not offer a learning system to learn and change over time. Therefore, a framework is proposed to address these shortcomings in the aim of partially automating the translation of the requirements into a software architecture which may potentially be adapted for distributed systems, coupled with a learning system.

4.2.1 Traditional approach

In a traditional approach, the typical steps of the SDLC to design and implement a large and complex software solution would include the requirements phase, the design phase, the implementation phase, the testing phase, the deployment phase and the support phase. It may seem mostly describing the waterfall methodology but even for other traditional approaches or more modern agile approaches, the requirements will have to be ready before the architectural design can be started. Figure 4.1 illustrates the key stages in the traditional approach.

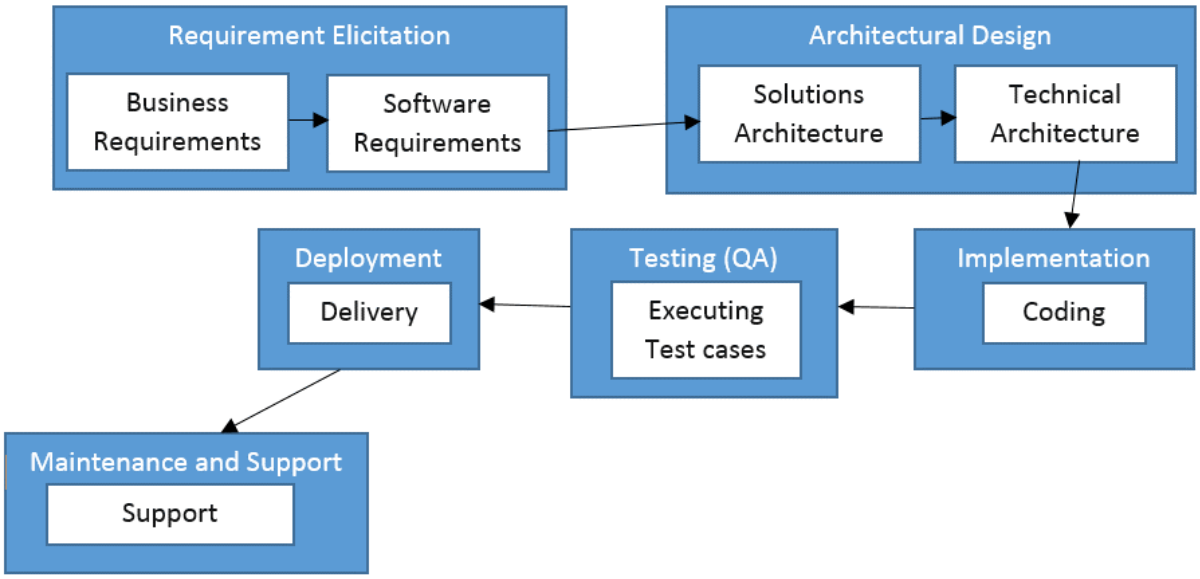


Figure 4.1: Traditional approach for developing complex software solutions

During the requirements phase, the client expresses the business needs which are collated as the business requirements. From these business requirements the software requirements are derived, describing what the software needs to accomplish in order to deliver on the business requirements. Afterwards the architectural design is elaborated, that is designing the solutions architecture, which can be considered as the architecture of the solution as a whole. The technical architecture defines the low-level architecture of the program. For example, the technical architect may opt for the MVC (Model View Controller) architecture at an application level which is usually when the development environment is selected. During the implementation phase, the solution is coded and then sent to the Quality Assurance (QA) team to be tested. When the solution is approved by the QA team, the solution is deployed and any issues are then handled by the maintenance and support team.

4.2.2 Proposed framework

As explained in chapter 2, the defects injected in the earlier phases of the software development life cycle (SDLC), which are carried over into the later phases can be the costliest to fix. Thus, one solution is to introduce automation in the early phases of the SDLC processes with the aim of preventing or minimize the introduction of defects. It is proposed to introduce automation at the requirements phase to automate the elaboration of design artefacts and derive an architecture.

Figure 4.2 (on the next page) illustrates how the proposed framework is laid out and the changes that it brings as compared to the traditional approach. The requirements are accepted by the framework and it is processed through the natural language processing (NLP) component so that meaningful concepts can be extracted from the text. These concepts are then passed on to the design extraction component where the use cases and the class diagrams are produced. The design extraction component is coupled with an ontology which uses a database to complement the contents extracted from the text. The output is then passed on to the abstraction engine which abstracts the design from the input of the text to a low-level design, a medium-level design and a high-level design. Simultaneously, the neural network is solicited to analyse historical data in the database to produce design elements and also assist in the abstraction of the design. The user would be required to verify or approve the output before it is passed on to the development team. The remaining phases of the SDLC are unchanged. The development team takes over and codes the application. The QA team then tests the application before it is deployed to the live environment. Afterwards it is the responsibility of the maintenance team to support the application once it is live.

The NLP component is first used on the requirements and it will clean and process the raw input so that it can be used for the ensuing steps. After the data has been processed by the NLP component, the data is then passed on to the design extraction component. The design extraction component will first extract the use cases. The actors of the system would be identified by the parser. There are various processing and abstraction steps required for the parser and the algorithm to identify the actors. When an actor is identified, the algorithm goes on to identify the actions, which then become use cases. The extraction of use cases and each of the individual components are further elaborated in the following sections.

The design extraction component starts by using the output of the NLP component in order to elaborate the class diagrams. Some of the key concepts identified are retained in order to identify class diagrams. The algorithm considers the potential of some of the actors as classes and then goes on to identify the methods and attributes. The output is then compared to the data stored in the ontology. The data from the ontology is used to enhance the output and the data found in the text is added to the ontology in case it is not present there.

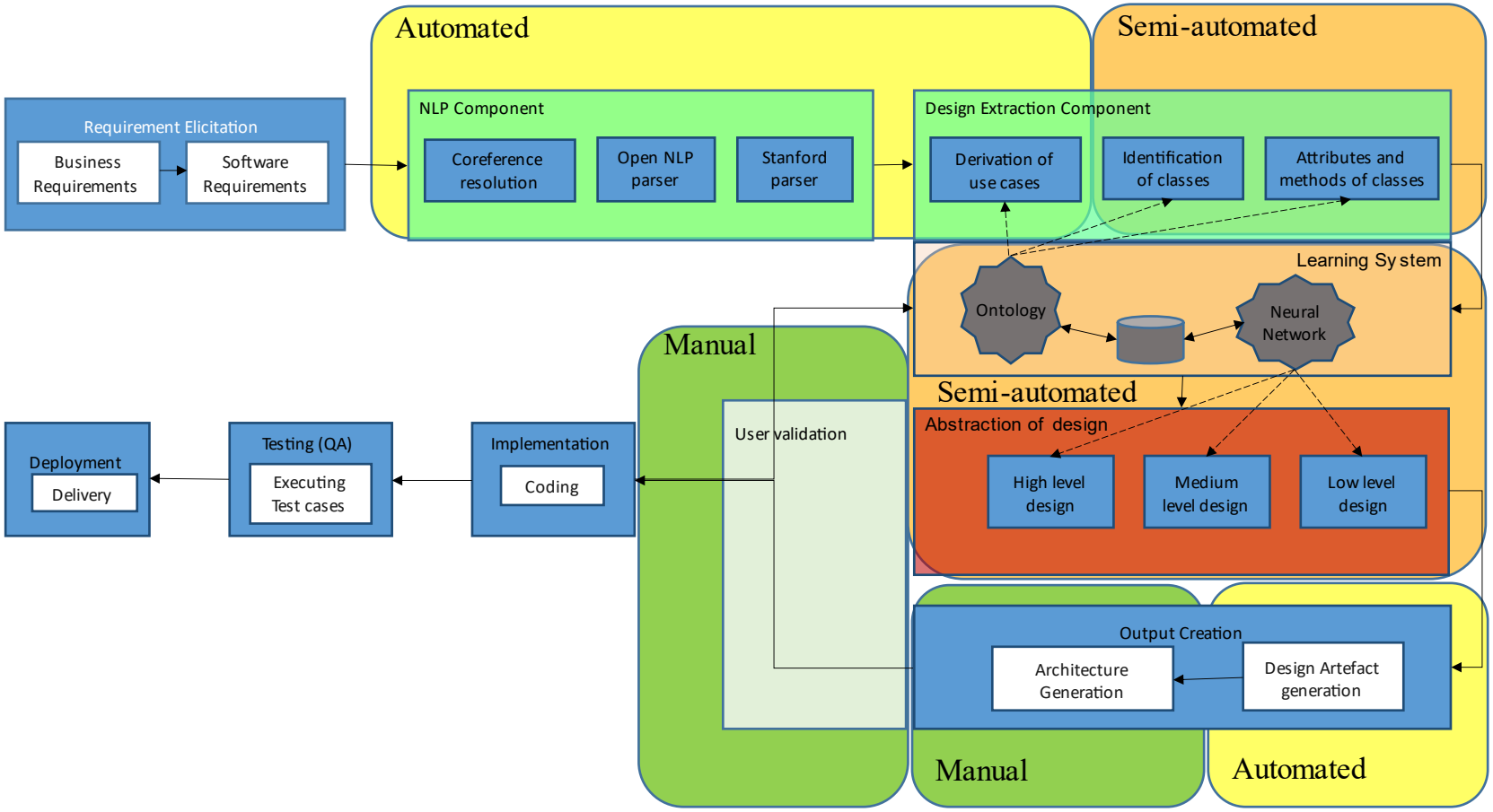


Figure 4.2: Proposed approach for designing distributed software solutions

All the steps against a yellow background are automated, all the steps/components against an orange background are semi-automated and those against a green background are manual tasks.

The output is an XML file which the user can edit, change and modify. The learning system is connected to a database, which stores the historical values of each design component in the backend. The list of these historical values is then used to predict the values to be added to the design for a given domain. The design created from the text is also enhanced and modified by the learning system so that a design in various levels of abstraction, can be created as an output.

The output from the framework can be changed by the solutions architect who validates the design artefacts outputted by the framework. The process has been partially automated and the technical architecture and the solution architecture are more likely going to be aligned, as they are modelled from the same requirement set. The models produced by the framework and the non-functional requirements can also be used to derive the test cases for the system. The remaining sections in this chapter provide a walk through the different components of the framework and outline how the concepts, principles and algorithms have been applied to achieve the whole framework.

4.2.3 Framework workflow

The framework would consist of several components, whereby each component would handle a particular element of the processing. The framework would adopt an iterative approach where the codes are parsed iteratively and the output from each iteration is passed on to the next for further refinement. The input would be requirements written in natural languages, namely English. The framework would constitute of several components, each of which would be performing specific tasks at various levels. The goal is to use the requirements as primary input and then parse, refine and clean this input so that meaningful concepts are extracted from the text. These concepts are then passed on to the subsequent stages so as to extract the UML artefacts. The ontology is solicited so that data stored in the ontology can also be used for derivation of unified modelling language (UML) artefacts. The user is then presented the results, whereby he/she would be allowed to make some changes (Figure 4.3). The final artefacts are then the output from the framework. At a high level the framework workflow can be represented as follows:

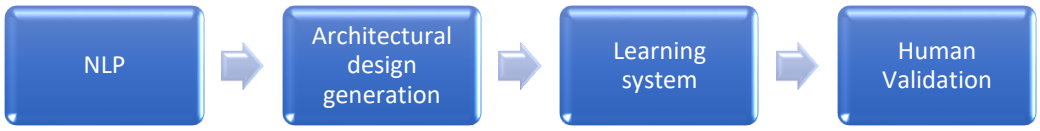


Figure 4.3: High level workflow

At a more detailed level, the main components can be illustrated in Figure 4.4:

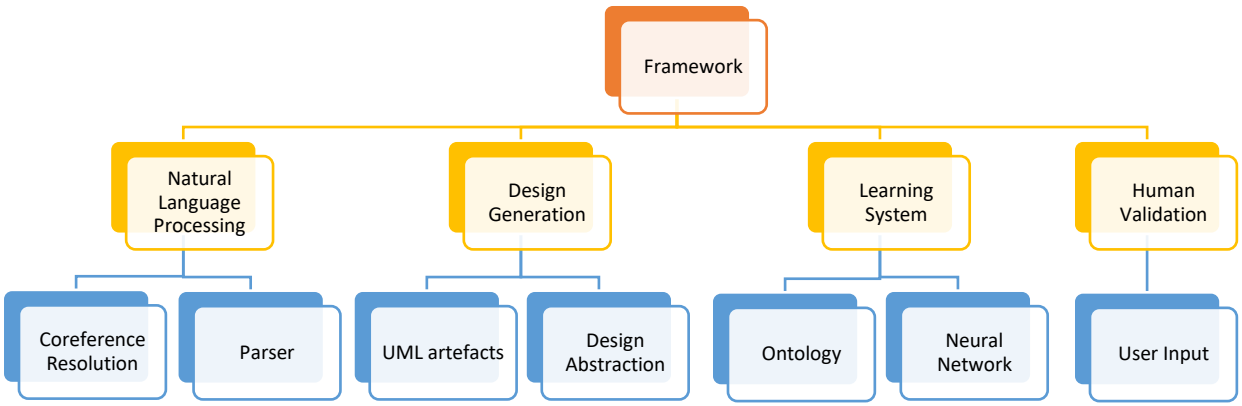


Figure 4.4: Sub-components of the framework

Each of the components are explained in detail in the following sections, with a brief explanation of how they work, how they interact with each other and how each component contributes to the output. Initially the user would have to upload a file to the system and select a "domain" for which the file is intended. The framework is designed to be domain driven, as the learning system and the ontology in particular are calibrated by domain. The system is not engineered as an open system but needs to be calibrated by domain as knowledge needs to be fed into the system so that design artefacts can be extracted for a specific domain. A domain can be defined as almost anything by the user but it is recommended that a domain be generic enough so as to encompass a certain field (activity / business line / service line) but should also not be too vague so that the knowledge to be represented is too bulky. For example, one domain for the whole of the insurance sector might be too bulky and the insurance should be split into sub-domains so that the ontology data can accurately represent the sub-domain and contribute to the output.

In order to extract meaningful information from the text, the requirement document is first taken up through the natural language processing component. The output of the NLP parser is used for further processing, in order to create the data structures required to extract UML artefacts, which are also enhanced with the help of an ontology. The output is also to be coupled with a learning

system, namely an ontology and a neural network. The output is then presented to the user who would be able to edit the output to best reflect the design desired. Finally, the output design artefacts should be generated by the framework. The major processing steps for the framework can be summarized as shown in Figure 4.5:

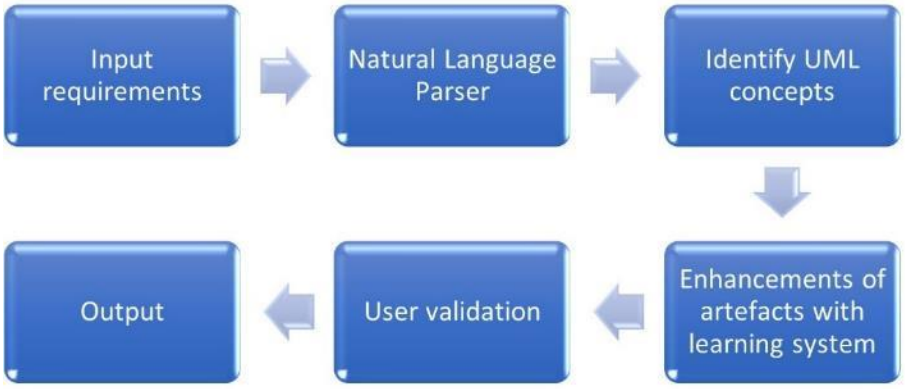


Figure 4.5: Main processing steps workflow

The six steps mentioned above can be regarded as the six major transformation on the data so that the user input is abstracted further and further. The following sections outline how the components are used at each stage to transform, abstract and enhance the output from each component so that the final output are design artefacts, desired by the user. The remainder of the chapter explains the key components, how they are used and main features of each the underlying and existing research which are crucial for the components to work. The next section starts by explaining how natural language processing (NLP) component works in general and how the NLP component of the framework was developed in order to deliver the key features needed by the framework.

4.3 Natural Language Processing (NLP) component

Natural language processing (NLP) is the area of software engineering which processes, parses and abstracts meaning of text into a data structure, so that it can be further processed. Languages such as English, German and French (etc.) are known as natural languages and the task of NLP is to parse and extract as much meaning as possible from the text and save it in data structures. These data structures can then be used for specific applications and / or can be combined with other areas of software engineering to achieve a definite purpose. In recent years, NLP has grown beyond natural languages and encompasses other areas of textual analysis. For example, there are NLP parsers which specialize in parsing of feeds from social media sites such as Twitter and Facebook

[119-123]. The language used on social media is not always structured with a proper grammar, but these platforms are immensely popular and a fantastic opportunity to gauge public opinion. The output from these parsers is very often used for sentiment analysis. For example, a mobile phone company can gauge the sentiment of the public by analysing feeds for a series of hashtags when launching a flagship product, usually after key product features and a product release date have been announced.

The more classic application of NLP is to analyse text, which are written in natural languages, which are properly formatted and edited so that all the grammatical and vocabulary rules of the language are respected. Informal language(s) and incomplete sentences are not expected and the natural language parsers are not designed to handle these situations. The applications of natural language processing vary greatly and include automatic summarization, machine translation, morphological transformation, discourse analysis, named entity recognition, sentiment analysis, speech recognition, word sense disambiguation and information retrieval. NLP has also been used with other programming concepts in order to enhance the user experience or improve product usability. For example, mobile operating systems now include speech recognition and the user can thus command the phone to perform certain tasks. The speech is translated to a textual format, which is then interpreted by an NLP parser and the results are passed on to the next component so that the phone can undertake the necessary actions to fulfil the command instructed by the user.

This research aims to derive design artefacts from natural language requirements. Therefore, the scope of the research does not investigate informal language structure such as social media comments and speech recognition. Whilst the research on NLP also include parsers and algorithms suitable for a series of languages, this research focuses on the English language. In order to further understand how NLP parsers and applications of NLP can help to assist in the process of requirement engineering, it is appropriate to analyse some of ways NLP parsers have been engineered to work. Two important and predominant concepts in NLP, which can be considered as building blocks, are part of speech tags and sentence parse tree.

There are many other concepts used in NLP, but the two above mentioned concepts help to make sense of a lot of the output from an NLP parser. They can be considered as the way developers and engineers have found to extract meaning from the raw data so that it is abstract enough but also avoids the trap of being excessively vague. Regardless of the meaning of the sentences, part-of-

speech tagging and sentence parse tree are used to represent the sentences of a language in terms of basic linguistic concepts (such as verbs, nouns, verb phrase, prepositional phrase, etc.). In this way the user can use a parser to make sense from any sentence and devise their custom algorithm on top on that for further processing. The next sections provide a short walk through of part of speech and sentence parse tree.

4.3.1 Part of speech tags

Part of speech tagging, also known as grammatical tagging or word-category disambiguation, can be regarded as the process of marking up a word in a corpus, based on its definition and its context. Typically, there is one tag for each word, bearing in mind that a word can be used in more than one manner. For example, words such as “book”, “account” or “store” may be used as either a noun or a verb, depending on the context of the sentence. Hence this exercise also considers the relationship of a word with adjacent and / or related word(s) in a paragraph, sentence or phrase. This task therefore pertains to the task of assigning a word a tag which is abstract enough to represent it in a context that it is used. In order to keep the output meaningful, most of the time the tags assigned correspond to a part of speech of the native language. For example, the tag “NN” is used to denote a singular noun. The process of part of speech tagging can represented in Figure 4.6:

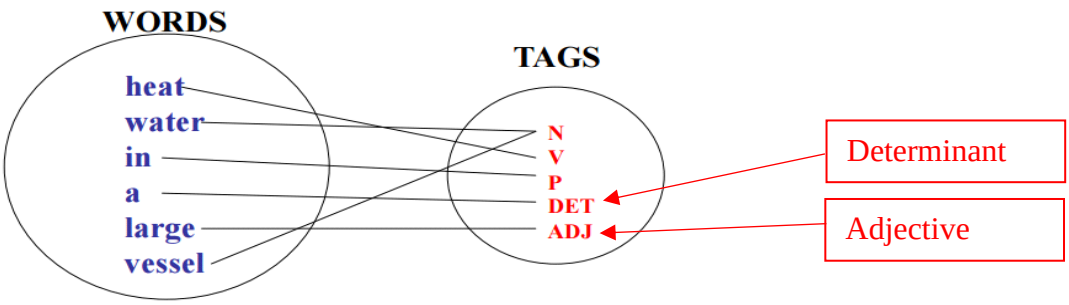


Figure 4.6: Graphical representation of Part of Speech tagging [124]

The tags used can literally be anything or any word. In computational linguistic, most taggers would allow the user to train the tagger on a tag set. The user may use any tag set that best suits his/her needs and the user may define a custom tag set. However, in order to keep the result meaningful and the output easy to use, the tags usually correspond to the category of word in the natural language. For example, for the English language, the ten common part of speech are: noun, verb, article, adjective, preposition, pronoun, adverb, conjunction, interjection and determinant.

Therefore, most tag sets for the English language make use of tag which correspond to the broad categories of part of speech. Please note that there may be more than one tag defined for a part of speech. The whole collection of tags is known as a tag set or a Treebank and there are many tag sets which have already been defined for various languages.

There are various Treebanks that are available for the English language [125] and the common and widely used part of speech tag set is the "Penn Treebank II tag set". It was developed at the University of Pennsylvania [126] and is frequently used as there is an adequate number of tags to make clear distinction between the concepts, while avoiding the trap of having too many tags which renders the exercise of interpreting the tags a tiring one. It contains 36 tags which denote the different part of speech and there are multiple tags to denote a part of speech. For example, there are 4 tags to express the different types of nouns which may be encountered in the text.

The tags listed in the Penn Treebank [126] set are listed in Table 4.1:

Number	Tag	Description		Number	Tag	Description
1	CC	Coordinating conjunction		19	PRP$	Possessive pronoun
2	CD	Cardinal number		20	RB	Adverb
3	DT	Determiner		21	RBR	Adverb, comparative
4	EX	Existential there		22	RBS	Adverb, superlative
5	FW	Foreign word		23	RP	Particle
6	IN	Preposition or subordinating conjunction		24	SYM	Symbol
7	JJ	Adjective		25	TO	to
8	JJR	Adjective, comparative		26	UH	Interjection
9	JJS	Adjective, superlative		27	VB	Verb, base form
10	LS	List item marker		28	VBD	Verb, past tense
11	MD	Modal		29	VBG	Verb, gerund or present participle
12	NN	Noun, singular or mass		30	VBN	Verb, past participle
13	NNS	Noun, plural		31	VBP	Verb, non-3rd person singular present
14	NNP	Proper noun, singular		32	VBZ	Verb, 3rd person singular present
15	NNPS	Proper noun, plural		33	WDT	Wh-determiner
16	PDT	Predeterminer		34	WP	Wh-pronoun
17	POS	Possessive ending		35	WP$	Possessive wh-pronoun
18	PRP	Personal pronoun		36	WRB	Wh-adverb

Table 4.1: Penn treebank II tag set

The 36 tags, along with the meaning assigned to each tag, are listed in Table 4.1. A simple example is shown below, whereby a sentence is shown before and after it is tagged. Each word is followed by a dash and the respective tag that is assigned to it. For example, the sentence "*The user must be able to buy travel insurance online.*" will be transformed to "*The-DT user-NN must-MD be-VB able-JJ to-TO buy-VB travel-NN insurance-NN online-NN .-.*".

Part of speech tags are therefore very useful to abstract the data to a set of standardized or agreed upon tags which can then allow the user to make more sense out of the output that may be produced by an NLP parser. A series of tag sets exist for the English language and there are also other tag sets for other languages. Part of speech tagging can be regarded as an agreed standard to abstract the data from the Natural Language to a data set which renders the words more easily accessible and readable. From this level of abstraction, it becomes easier to use the words as concepts and extract meaning from the raw data.

The user can then use these part of speech tags as a starting point and from there derive their own algorithm to make sense out of the data. The user can use the result from the part of speech tagger in an algorithm in order to extract meaningful concepts and manipulate the data in such a way that the output can be used for the required purpose. Part of speech tags is a good starting point to abstract the data and start making sense out of it, from a computational linguistic perspective. Part of speech tagging may be used on its own or may be combined with other concepts of NLP. Part of speech tagging is also used in building the parse tree and the next section explains the work and application of parse trees.

4.3.2 Sentence parse tree

A parse tree is a data structure very often used to represent data that is being parsed. This is not only limited to natural language processing as parse trees are derived for all sorts of processing. Figure 4.7 is an example of the parsing of an arithmetic operation.

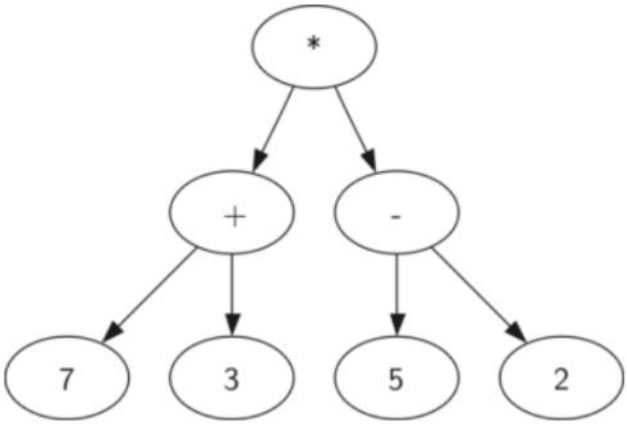


Figure 4.7: Parse Tree for an arithmetic operation [127]

In the context of natural language processing, a parse tree breaks down a sentence into a series of noun phrases, verb phrases, prepositional phrases or other sentences. The following is an example of the parse tree for a sentence, as shown in Figure 4.8.

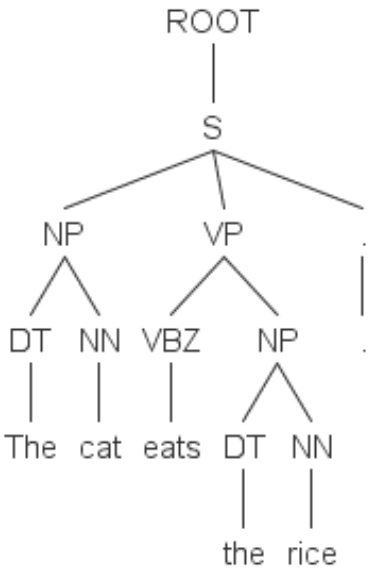


Figure 4.8: Sentence Parse Tree for a simple sentence

The root sentence, “The cat eats the rice”, is separated into a noun phrase and a verb phrase. From there each word is assigned a part of speech tag, until eventually the whole sentence is read. In order to come up with a complete sentence parse tree, X-Bar theory is used [128]. In a nutshell, the key word for each phrase, also known as the head word, is determined and the phrase is determined based on the head word. For example, the head word for a verb phrase must be a verb and a noun must be the head word for a noun phrase. The rest of the sentence parse tree comprises

of specifiers and complements. A specifier is the sister node of the head word tag and the complements are the children nodes of a head word tag. The tree is constructed such that the order of the bottom most layer always corresponds to the order of the words as they appear in the sentence.

A sentence may have more than one parse tree. As the size of a sentence starts to grow, the exercise of determining the sentence parse tree also becomes more complex. If the sentence also starts to become ambiguous, then the sentence parse becomes harder to derive. The parsing algorithm would still determine a phrase based on the head word and when more than one word can be used as a head, then there is more than one possible parse tree. This is known as structural ambiguity. Furthermore, there are different approaches and algorithms which can be used to determine a parse tree. Therefore, the exercise is not a straight forward one, but it is an important step of representing a sentence as a tree which can then be interpreted to fit in an algorithm. Thus, the parse tree is also another common and regularly used data structure in NLP.

Some of the underlying concepts have been briefly expounded. From there on parsers are built and there are various types of parsers such as context free grammar parsing, probabilistic and statistical parsing, dependency parsing and constituency parsing. The algorithm used in a parser also vary from parser to parser and can yield different results, which means that two context free parsers may not yield the same parsing result and accuracy for the same corpus. On top of that other enhancing or learning capabilities are added and more and more parsers are incorporating a learning capacity like neural networks in their parsing algorithm. Developers then use these parsers and can use the data structure created by these application program interfaces (API's) in order to achieve their needs.

There are many more concepts, data structures and algorithms which affect the work around NLP but the above-mentioned concepts can be regarded as the key building blocks which give an insight in the work of NLP. The main focus of this is to use NLP parsers, taggers and algorithms to derive meaningful concepts from the text so that it can then be used to produce a design for the architecture of a software based on the requirements. One of the applications of NLP is coreference resolution, which is the identification of the same entity in the text when it is referred to by different words. The next section elaborates on coreference resolution in details.

4.3.3 Coreference resolution

Coreference occurs when two or more expressions in a text refer to the same thing or person. A very simple example could be the sentence: "*Bill said that he would come*." In this sentence, both the words "*Bill*" and "*he*" are referring at the same entity (person). When the coreference occurs within the same sentence, it is quite easy to follow and keep track of the entities and their referents. However, when the multiple reference start to spread out in the text, it starts to become more complicated to keep track of the words pointing to the same thing or person. The area of NLP which deals with the identification of the same concept referred to by various words in the text is called coreference resolution.

In computational linguistics, the task of coreference resolution is achieved through an algorithm which is devised to identify all the words or expressions, scattered across the corpus, which refer to the same concept. The algorithms are designed to work even when concepts or reference to a concept span over more than one sentence or even a paragraph. The task is harder than it sounds as it is not confined to common pronouns. As shown in the example, names such as "*Bill*", also have to be identified and these names are gender specific. The task can be very valuable in computational linguistics as it is a way to make the computer or the machine understand the link between words pointing to the same concept.

This aspect of computational linguistics is valuable to the proposed framework as it would allow the NLP component to reduce the number of redundant concepts. The concepts, subjects and objects used in a sentence could be grouped so that there is a minimum number of concepts. Thus, the design proposed would not be clogged or overloaded with very similar concepts which are in fact pointing to the same thing. Coreference resolution is an important part of NLP but the main task is to parse the sentences and tag the words. The next sections describe how this part of NLP is used with respect to the other components and how the design is then elaborated.

4.3.4 Parsers and taggers

As discussed before, the first step to process the requirements is through an NLP component. Before the text can be processed, it is necessary to clean the text. The first step is to eliminate special characters. Whenever special characters, which are not directly part of the requirements are processed, the parser can fail to parse a sentence or parse it inaccurately. Special characters

would include bullet points or other characters used to enhance the readability, but do not contribute to the requirements. The resulting text is then used for the next stage which is the isolation of sentences. Whenever an input file or corpus is uploaded by the user, it is important that each sentence be processed individually. In case the sentences are not processed individually, there may be concepts from one sentence mixed with concepts from another sentence.

The third step is to resolve the coreference so that the number of distinct words and concepts can be reduced to a minimum. The reason for this step is to avoid many generic concepts in the design. Coreference resolution would be done programmatically through an online library (or parser) which offers this capacity. The aim is to use the library to replace as many words as possible by the actual concept that they are referring to. In order to prevent this exercise from replacing very similar and accurately defined terms, the replacement of terms is confined to proper nouns.

The fourth step is the reduction of concepts. Basically, similar concepts or synonyms for the same word can be replaced with the same term so that similar and interchangeable terms do not clog up the design, which may lead to an inaccurate design or confusion. The fifth step is to parse the requirements, which have now been pre-processed by the first four steps, so that meaningful concepts can be extracted from the text. The final step is to use the data from the parser and custom processing rules to store the concepts in data structures which are then later used. These are the main steps in the NLP component.

The processing steps of the NLP component of the framework:

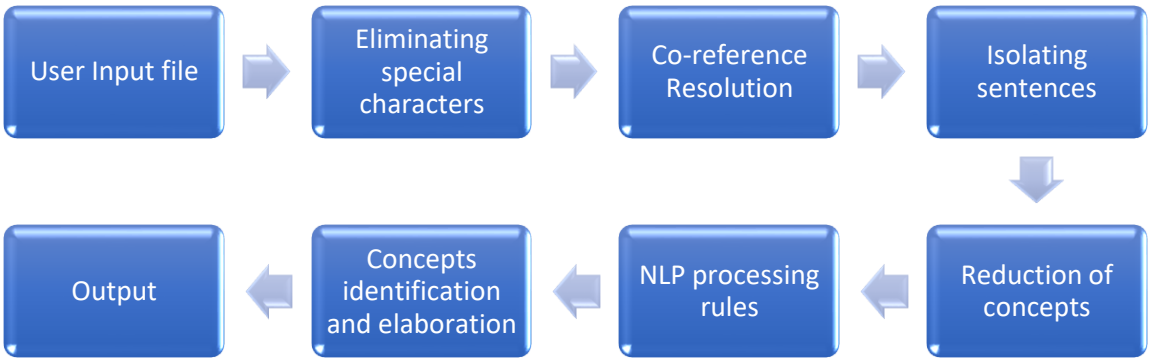


Figure 4.9: Processing through NLP component

Figure 4.9 demonstrates all the transformation steps carried out on the input from the user and there are six main transformation steps. Each of the transformation step in the NLP component

would either use a readily available library or a custom logic. The following sections describe more in detail how the NLP capacity were used and customized the transformation of the text through the NLP component.

4.3.5 Application of NLP to the framework

As outlined in the literature review there are many existing algorithms, frameworks and tools which use the capacity of NLP to process requirements. The algorithms can be elaborate and make use of a lot of existing research on NLP and other areas of software engineering. Inspired by these researches, it was decided that the algorithm for the framework would have an iterative approach. The raw text could be filtered, cleaned and enhanced in phases, with each phase using the output from the previous phase. In this way, the text is progressively cleaned and made more adaptable for concept extraction and generate the artefacts. The approach is similar to code slicing [129], except for the following differences:

- There is no code involved and the input is text in a natural language (English language is expected).
- During code slicing, the source code is summarized and occasionally redundant elements and portions are removed. With the text, there should not be any redundant text and removing text should not lead to the loss of substance.
- The meaning of the text is not simplified or abstracted.
- Each phase (slice) is meant to clean or enhance the text.
- Words which are similar, comparable and interchangeable are reduced to a common word or base form so that the overhead of processing is reduced.

Having explained some of the concepts of NLP and how slicing is intended to be applied to the framework the next sections explain how the NLP component of the framework is created. As shown in figure 4.9, there are six main stages in the NLP component which transforms the text and extracts the key concepts to be used for the elaboration of the design. The stages occur sequentially and the output from each stage is used as the input to the next.

4.3.5.1 Eliminating special characters

In order to avoid special characters or unknown characters in the text, this step is devised to reduce the number of such type of characters. The reason for doing this is to avoid these characters while the parser is used as it may yield inaccurate results. The special characters or unknown characters are not the common characters usually encountered in the text such as the following: ! @ # $ % ^ & * ( ) . The characters can be bullet points or special characters which may be present when the parser is attempting a file from a file format which is not a text file. The user is advised to load a text file with a ".txt" extension, but in case other file formats (even common ones such as a word document with ".doc" or a ".docx" extension), there may be special characters which are read by the parser. For this reason, this step is intended to eliminate special characters which may cause the parser to go off course.

4.3.5.2 Coreference resolution

Before the input can be parsed by the framework, there needs to be some pre-processing in order to prepare the text. The pre-processing is a cleaning task which is meant to prepare the text for further processing, while preserving the substance of the text. The main step is coreference resolution and it is meant to identify the words which are referenced by other words. Words, usually nouns, can be referred by other words, usually common pronouns. In order to avoid common pronouns in the text and also reduce the possibility of having redundant concepts in the design, the reference of certain meaningful words by other generic words, coreference resolution is applied. The goal here is to identify and replace generic words by the key words they are pointing to. After this stage, text is altered and this altered version is then used for the next step.

4.3.5.3 Isolating sentences

After coreference resolution, all the individual sentences need to be isolated. This step is needed to prevent the parser from parsing a paragraph of multiple sentences in one go, but rather process one sentence at a time. The accuracy of the parser is better when a sentence is processed one at a time as opposed to a paragraph of sentences. A parser or library with that capacity would be used to achieve this step, so that built-in functions can readily process the text and identify the sentences. This step is carried out so that the text can be parsed accurately later on.

4.3.5.4 Reduction of concepts

After the common pronouns have been reduced and the maximum number of words are represented by the distinct word which is not a common pronoun, the next step is to reduce the number of concepts. A text may contain several words which represent only one relevant entity. For example, the words “clients”, “customer” and “user” may correspond to a one and the same person or role interacting with a software system. In order to avoid these three words to be represented by three distinct entities in the design, these concepts need to be represented by one word or a root concept. Therefore, the following rules are devised to reduce the number of concepts.

- All the words in plural can be reduced to their singular form.
  This would prevent the concept from being picked up more than once. For example, “customers” and “customer” should not really appear in the design as two separate concepts.
- Irregular plurals are listed in a database table. All irregular plurals are replaced by the singular word form.
- A synonyms table is also expected to be used so that similar and comparable words can be grouped as one term.
- The synonyms table can be used to reduce concepts in the sense that the plural form of a word can be replaced by the singular form of the word.
- In case a concept is changed, then this table can also be used. For example, in case it is decided that there should be only one word like “customer” of all user interaction, then the word “users” can be replaced by the word “customer”.

This step avoids leaving very similar, comparable and interchangeable words as distinct concepts. This could result in a cumbersome design whereby almost identical elements are repeated and clog the design, making many concepts redundant.

4.3.5.5 NLP Processing rules for the corpus

Inspired by the literature review, a series of rules have been elaborated in order to process the text, so that meaningful concepts are identified. This set of steps also consolidates on the previous step and builds a series of word lists, which are of interest. These lists are kept and used in the subsequent steps for concept extraction and eventually generation of artefacts.

- Identify the last word of each sentence. Save these words in the list Stop words.
- Keep track of the total number of words in the text. Also keep a running count of the number of occurrences of each word.
- Calculate the frequency of each word.
- Use an NLP (Stanford) parser to parse the whole document.
- Store all the nouns in a list, the Nouns list.
- Store all the verbs in another list.
- All the nouns are saved in the concepts list.
- All the subjects of the verbs (regardless of whether they are nouns) are also added to the concepts list.

These rules build several lists, which can then be used in the subsequent steps to group the concepts and slowly build the artefacts. From then on, all this data is stored in the database and data structures such as lists or custom objects.

4.3.5.6 Concepts identification and elaboration

The next significant step is to produce UML artefacts from the text. Before this can be achieved, it is important to consolidate on the steps of the NLP components described above and identify the key concepts in the text. These concepts are then grouped, collated, analysed and used in the derivation of the design elements of the architecture. The concepts which have been retained so far, in the Concepts List, are going to be used for the elaboration of use cases and derivation of class diagrams.

- All the concepts (coming from the noun list) which have a frequency of less than 2 % are ignored for future processing.
- All the subjects of the verbs are considered for future processing, regardless of frequency.
- Words which are too generic, or correspond to a design element, a proper name or a geographical location are ignored for future processing. The database stores a list of words which may be ignored as they are irrelevant or too generic. For example, words like

"machine", "function", "London", "John", etc… can be added to that table so that they are ignored.

After all of the above-mentioned steps, the output from the NLP component is a list of concepts retained for the elaboration of the design. The next step is to use concepts to elaborate the design.

4.4 Design extraction component

All the previous steps have been preparing the raw text in order to extract key concepts from the requirements. The list of concepts and lists of words, data structures and processing rules shall then be put to use so that an architectural design can be obtained. Firstly, the design extraction component aims to obtain a use case diagram and then a class diagram. The next section exposes how the output from previous sections are used in order to produce the design artefacts. The output from the NLP is validated through a set of rules (described in section 4.4.1 and 4.4.2) and only the concepts which are relevant or pass other rules (like exceed 2% frequency in the corpus) are placed in data-structures to be used for the design.

4.4.1 Deriving use cases

The output from the previous sections, the multiple lists of concepts and the following list of rules outline how a series of use cases can be derived.

- From the output of the parser, the subjects of all the verbs are potential actors.
- If the subject of the verb is a valid concept (passes all the validation rules above), then that concept is used as an actor.
- For each sentence which contains a valid actor, the tense of the verb is checked. Present tense is the default expected tense. Other verb tenses such as the past tense or the past participle are ignored.
- The rest of the sentence, starting from the verb onwards, is considered as the action part of the use case.
- For each sentence, the use cases are then filtered, using the following rules.

- All the use cases derived for a sentence, using the rules above whereby the valid concepts are the actors and the rest of the sentence, starting from the verb are use cases (actions).
- Duplicate or similar use cases derived from the same sentence are eliminated.
- The verb and its direct object are retained for further processing. It would be used for the elaboration of a class diagram.

4.4.2 Deriving class diagrams

In order to extract class diagrams, the work achieved by the NLP component and the work for the derivation of the use cases are used. Class diagrams are then created by using the following rules:

- All the actors are considered as classes by default.
- All the verbs and direct objects of these verbs are considered as a method for that class.
- The text is then scanned again. If there are two consecutive nouns, and the first one is a class then the second noun is considered as an attribute of that class.

The above sections conclude the way the framework is designed to work at a high level. All the steps used so far have used the input of the user, libraries to reduce concepts, replacement values from the database. All the above steps have the aim of filtering and cleaning but with the main goal to retain the substance of the input text. The next step is to use external data or historical, so that the framework can enhance the output produced so far. The goal is to use a learning system which comprises of an ontology and a neural network to complement the output generated so far.

4.5 Learning system

Learning system can be a vague term to describe the ability of software system or a machine to learn, adapt and grow. Therefore, it is important to explain how the learning capacity intended for the framework works. It is planned that the framework would have some capacity to go through expert data pre-entered for a domain and propose or suggest possible design artefacts as an output. This is in contrast with the existing approaches which aim to derive a design from the requirements using classic NLP techniques. It was decided that a learning capacity should be added in order to allow the software the ability to somehow make different decisions, learn and improve or alter the

possible outcomes when processing an input file. Whilst the aim is not to achieve the very latest deep learning algorithms and autonomous learning artificial intelligence, it was still deemed necessary to have some learning capacity that the framework can rely on. This section explains how the learning system is used in the framework, the potential gains and which aspect are affected by it. There are two main learning strategies in the framework and they are

1. the ontology and
2. the use of neural network to assist in the elaboration and abstraction of the design.

The ontology is intended as a means for the framework to consult expert data pre-entered or custom data entered by the user in order to propose a design for a given domain.

4.5.1 Ontology

"Ontology" is a word first used in the study of philosophy and the elaboration of philosophical thoughts and currents of thoughts. Ontology, in its classical sense, can be regarded as a study of "what there is" or the thoughtful process of defining the nature of being, becoming, existence and/or reality. It dates back to the ancient Greek philosophers, and renowned philosophers such as Plato have been known to participate in the work on ontology and philosophy in general. Ontology has thus been part of philosophy since the Greek philosophers.

With regards to computer science the word ontology is different as it pertains more to the representation of data and concepts and how they can be used. Although inspired from the traditional work on ontology, the research on ontology in computer science is far less philosophical and has the main purposes to document and gather knowledge about the relatedness of concepts, such that it can be used for commercial applications. The first definition of ontology, as it is applied in computer science, can be attributed to Gruber [130] and it goes as follows: "An ontology is an explicit specification of a conceptualization", which is based on the definition of "conceptualization" as provided by Genesereth and Nilsson [131], whereby it can be "the objects, concepts, and other entities that are presumed to exist in some area of interest and the relationships that hold them".

The work on ontology has been growing [132][133] and there have been many researchers who have contributed on various aspects of ontology. A subfield of artificial intelligence is knowledge representation and the research on ontology has helped to improve the results of knowledge

representation. The work of knowledge representation and ontology became more and more important with the adoption of the world wide web [134]. With the growing use of the world wide web, and the mass usage of search engine, researchers started to work on algorithms which could compute the “coefficient of relatedness” between terms and how to represent this information [135]. This area of research is often referred to as the semantic web and the research on ontology has assisted the development of a manner to compute the relatedness between terms.

Having said that, there is no definite or standard way to define or use an ontology. Any user is free to design an ontology in any fashion that he / she likes. In order to design an effective ontology, the purpose of the ontology needs to be clear so that the design would be efficient, purposeful and easy to maintain and scale. For example, an ontology would have to show relatedness between terms, must be applicable to that domain, and there must be provision to scale the ontology if need be. An ontology having to show relationship between terms which may be hierarchical would be different from an ontology having to show relationship between non-hierarchical terms.

There are also certain tools which may be used to produce an ontology. For example, researchers at Stanford University have used the Protégé software to generate an ontology [136]. For the sake of explaining how Protégé works, the authors describe how to create an ontology for the classification of food. The advantages and uses of defining an ontology are explained and the article continues to describe how an ontology could be defined, using the Protégé software. The ontology which is shown is a hierarchical one, used to classify and organize the different types of food (Figure 4.10).

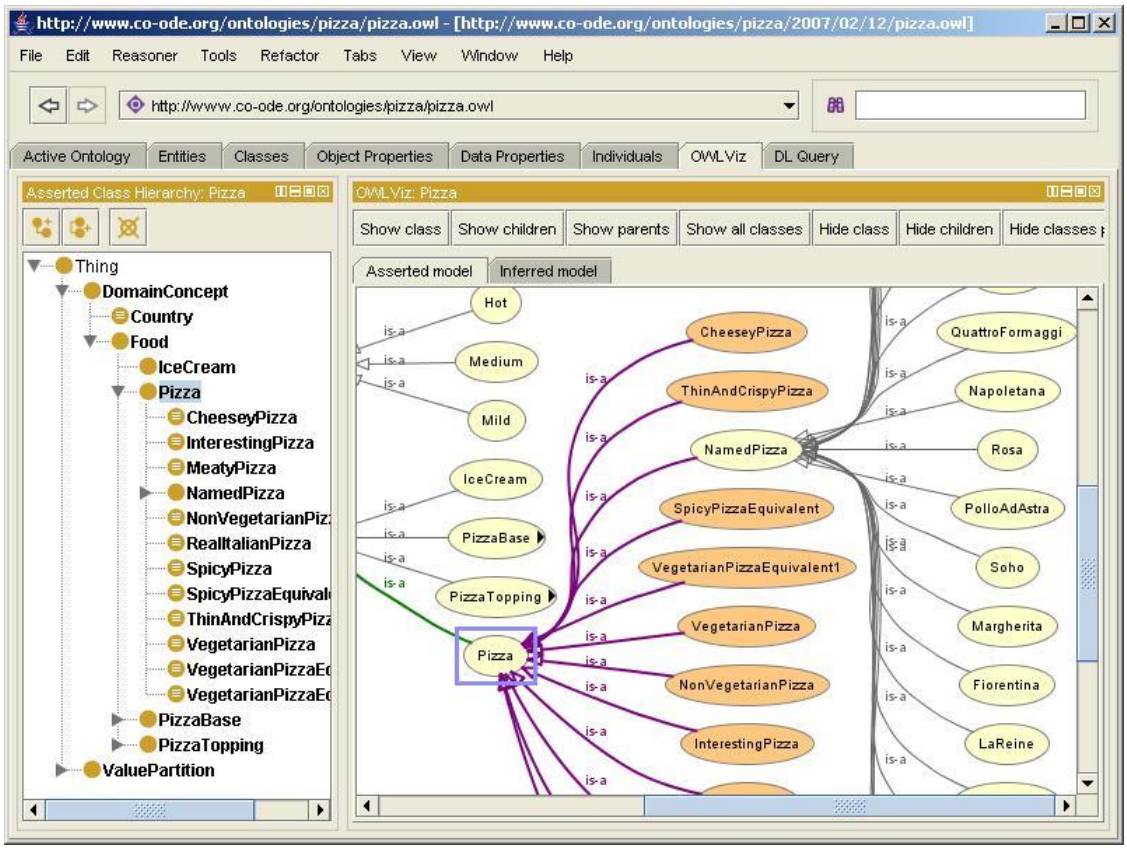


Figure 4.4: Hierarchical ontology in Protégé [137]

The example used is a hierarchical ontology, whereby most of the terms and concepts are defined as a sub class of a broader concept. One root concept is “Pizza” and the next children concepts are “CheeseyPizza”, “InterestingPizza”, “MeatyPizza”, etc.... The ontology goes on to enumerate the various types of food, classified and listed with respect to a parent node.

Ontologies are also used to represent data which are not hierarchical. For example, given a term like “Fanta”, an ontology could be used to represent how other terms like “Sprite”, “orange” or “soft drink” are related to that one term. In this way, knowledge can be gathered as to whether another branded soft drink, the colour and taste of the product or the general category of a “soft drink” is more related to the term “Fanta”. This can in turn help to assist the algorithms in search engines to filter or propose concepts to the end user. This use of an ontology is more common and is used in semantic web [138].

The Web Ontology Language OWL is the ontology standard recommended by the W3C (World Wide Web Consortium) [139] for creating and using ontology for the world wide web. The

standard has been revised and OWL 2 [140][141] is now the recommended standard for devising ontologies for the web. These standards are defined so that ontologies for the world wide web are accessible and understandable between a large number of users and providers. Therefore, the ontology is more valuable as it can be used by a large number of stakeholders and the transition towards semantic web is smoother and faster. This in turn helps search engines to be more accurate and cover a larger set of websites in the result set.

The work for this framework is not exclusively on ontology and is not intended to be used for semantic web. Therefore, a much simpler and more applicable ontology, is to be designed. The aim of the ontology is to track the relatedness between the terms and some of the relations between the terms would be hierarchical. The aim is mainly to get an architecture from the text and therefore the relevant design elements are to be represented in the ontology. There are certain design elements which may be hierarchical. For example, the attributes and methods of a class would not be on the same hierarchical level as a class. The ontology would show the relatedness between the classes and the hierarchy between a class and its attributes and methods.

In order to define an ontology which shows the relatedness between terms and the relationships between some of terms in a hierarchical way, it is important to organize the data so that a partially hierarchical relationship can be created and used. Therefore, the ontology should be able to compute a "relatedness index" or simply an index which shows how two or more terms are related. It is also important that the weight is reinforced in case the relatedness between two terms appear more often in the text or is weakened if the two terms do not seem to be related as initially thought. In so doing the ontology behaves like semantic web where the coefficient of relatedness between two terms is reinforced or weakened at each round of processing.

The objective is to design a simple ontology, which is pre-populated with default design elements which may be partially hierarchical. For example, the relationship between a class and its attributes and methods would be hierarchical and the relationship between classes would not. The ontology should then reinforce the knowledge in the ontology in case, the same knowledge is encountered in the text. In case the pre-populated knowledge is not relevant to the domain, then the knowledge in the ontology is weakened. In case knowledge present in the text is not present in the ontology, then that piece of knowledge needs to be added. Therefore, as more and more documents are processed through the framework, the ontology dynamically adds new terms, consolidates existing

values and weakens less frequently encountered terms. As a result, the ontology is always learning by computing a weight which helps to add relevant terms and eliminate redundant terms. The next section describes how the weight system is implemented.

4.5.1.1 Using weightage in an ontology

An ontology may compute the coefficient of relatedness between two terms by calculating a "weight" or assigning a "weightage" between two values. The calculations or this way of representing the data has been inspired from neural networks and the semantic web. The aim is to use a weight, which is a computed numerical value to somehow show how one or many concepts can be related to each other. The value of the assigned weight itself can also be used to show the significance or relationships between two concepts. Since it is a numerical value, the higher the value the stronger the relationship between the two concepts and vice versa. The inspiration from neural network is one of the ways the weights are used in ontologies to depict the "relatedness" between concepts.

The work on neural networks is a field of its own and it is constantly evolving, with the adoption of deep learning algorithms. The basic principle behind neural networks is to mimic a human brain with a series of nodes which are meant to emulate the synapses in the human brain. Neural networks can start with a few nodes (neurons), spread across at least three layers (an input layer, a hidden or intermediate layer and an output layer). Deep learning algorithms have more elaborate configurations. There are different algorithms to adjust the weights between two neurons. There are back propagation algorithms which re-adjust the weights of each node retroactively. The following diagram illustrates how a neural network can be visually represented and how the weights exist between the two sets of neurons, as shown in Figure 4.11.

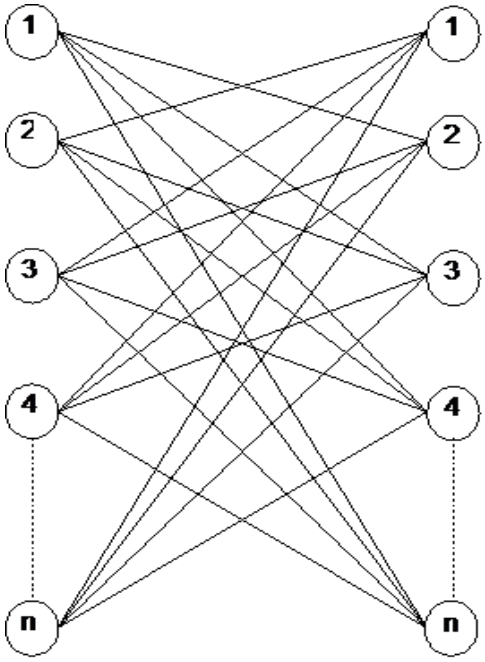


Figure 4.11: Inter relationships in nodes of a neural network

Each line connecting two nodes above represent a weight between the neurons. There are various ways to have an algorithm that updates that weight. The algorithm elaborated for the framework is inspired by the work done in semantic web. The aim is to define two components, a user defined component which the user can edit and a system defined component which is dynamically adjusted by the framework. The following algorithms provides a walkthrough of how a consolidated weight can be derived from two components.

The following formula is used to adjust the weight between two nodes.

$W = R \times W_{system} + (1 - R) \times W_{user}$

Where W is the total weight that defines the strength of the link between two entities,

W_{system} is the weight assigned by the system. The values are pre-defined to start with and the framework gradually maintains the weight thereon.

W_{user} is the weight defined by the user, who is free to alter this value at any time.

$R \in [0,1]$ is the ratio of the system weight and the user weight.

Thus, when

R = 0, the total weight is uniquely defined by the user defined weight

R=1, the total weight is uniquely defined by the system defined weight

Any other values of *R* imply that both system weight and user weight contribute toward total weight.

The user can configure the value of "R" which is a ratio which determines how much of the system weight or the user weight needs affects the final weight. The resulting weight is then compared to a threshold and the design element from the ontology is added to the design extracted by the NLP parser based if the final computed weight exceeds the threshold. All the design elements extracted by the NLP parser are then compared to the ontology and the terms present are consolidated by having the weights increased, the newly discovered terms are then added to the ontology with a default weight. The terms present in the ontology but not present in the text or not discovered by the NLP parser, have their weights reduced. Thus, the ontology is always adjusting the weights so that the data in the ontology reflects the terms frequently encountered by the domain. It is also expected that there would be terms which are not present in the text, but would have to be included in the design. They are known as default terms and the ontology also keeps track of those, whereby the weights then do not affect such terms. For example, a bank account would have to have an attribute to store the account number, regardless of whether that is mentioned in the text.

4.5.1.2 Use of ontology in the framework

As mentioned earlier, the ontology is designed to contain the data grouped by domains. Therefore, in order for the user to ensure that the output is enhanced with the ontology, the domain would need to be set up in the database. At the highest level, the domain is stored. For the ontology to effectively and efficiently assist in the derivation of the design elements, the right domain has to be set up and the information present must be relevant and purposeful. The information in the ontology will then be dynamically assessed and updated. The way the ontology works was covered in details in section 4.5.1.

4.5.1.3 Data for class diagrams

For each domain, a series of classes, methods and attributes are defined. For each entry there are two weight components defined and they are known as the "user defined weight" and the "system weight". To start with, default values are assigned to these weights. As and when the classes

defined in the ontology are encountered by the parser, the "system weight" of that entry is reinforced. The opposite also applies, since when an entry is present in the ontology but is not retrieved by the parser as a class, the system weight for that entry is reduced. The "user defined weight" has a fixed value which can be updated by the user. The resulting weight is a value between 0 and 1, which is then compared to a threshold value. The threshold value can then be used to determine if the class, method or attribute is accepted in the design or not. The threshold value can also be decided by the user.

4.5.1.4 Attributes and methods

For each class, there are a series of attributes and methods which are defined. For each attribute, there is a flag defined to determine if the attribute is a default one. Usually, a class would have a unique identification field which may not be explicitly mentioned in the text, and the default attribute flag is used to represent that. When an attribute is defined as a default attribute, that attribute is directly included in the design, regardless of the weights assigned. These are considered as mandatory identification fields which are needed in the design. For example, each class "user" may have a default attribute of "User ID", without it being mentioned in the text.

Similarly, each attribute is assigned two weight components, a system weight and a user defined weight. Just like for the classes, the system weight is adjusted based on whether the attribute is encountered in the text or not. The user can update the user defined weight at any time, in the database. The resulting weight is calculated based on the same formula and is accepted if the resulting weight exceeds the threshold value. The same principles have been used to define the class methods. For each class, there are a series of methods. The ontology is designed to work in the same way for class methods as it does for class attributes. There is the possibility to define default methods which need to be present for a specific class. Apart from this, there are also the methods which are defined in the ontology with a system weight and a user defined weight. Similarly, the system weights are strengthened if retained by the parser and the weights are weakened if not found by the parser. The ontology can also be updated with new content which are encountered in the text, but are not present in the database. Over time the weights of these newly added elements are also adjusted. For each value that is computed, the result needs to be compared to a threshold value. The user can edit that threshold value and thus decide on the input data proposed by the ontology. A high threshold value would mean that the few concepts from the

ontology would be proposed and a low value would mean that more concepts from the ontology could be retained.

The second part of the learning system for the framework is the neural network. It is intended that the neural network would have two features, the first would be to produce design elements, similar to the way the ontology and secondly to assist in the abstraction of the design.

4.5.2 Neural networks

Before explaining how the neural is applied to the framework, a brief overview of neural networks, and how they function is provided. The work on artificial neural networks began in the late 1940's [142] and started adopting current features attributed to it in the 1960's [143]. Artificial neural networks are inspired by the neural structure of the brain and are relatively crude electronic models which attempt to mimic the functioning of the brain. The brain learns from experience and artificial neural networks attempt to make use of this learning ability in order to recognise patterns and solve computational problems. The process involves the creation of massively parallel networks and training those networks to solve specific problems. The brain uses a network of neurons, whereby a biological neuron may be connected to as many as 200 000 other neurons, although a neuron has connections from ranging from 1000 to 10 000 other neurons. An artificial neuron copies a few of the characteristics of the biological neuron, and the key characteristics are expanded below. The next diagram shows a graphical representation of an artificial neuron.

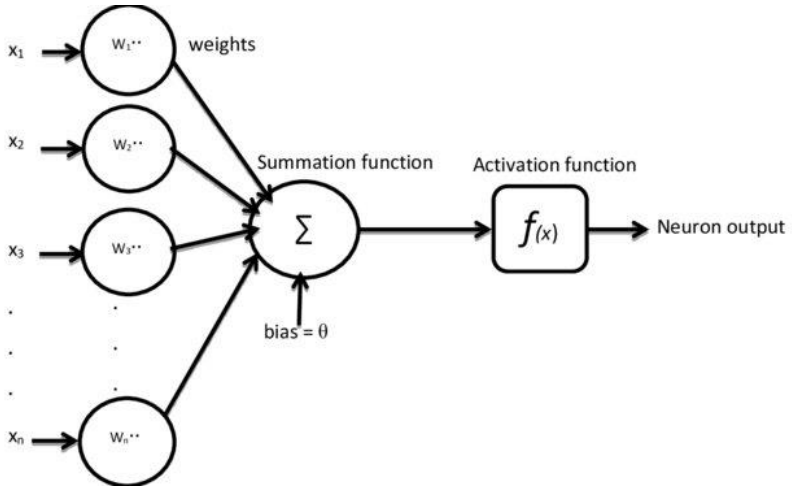


Figure 4.12: An artificial neuron [169]

In its simplest form, a neuron has a series of inputs, which may be weighted connections with other neurons or inputs to the neural network, a computation function and an output. An artificial neuron is more complex and the key features of an artificial neuron are expanded below:

1. Weighting factors.

   Each neuron may have a series of inputs and each of the input is associated with a weighted connection. The weight of the connection determines the significance of the input and is used to compute the output value of the artificial neuron. The weight is adjusted during the training process so that the contribution of the neuron helps to yield the expected result.

2. Summation Function.

   The summation function's purpose is to amalgamate all the inputs and their corresponding weighted connections so that the artificial neuron can compute an output. The simplest summation function is the dot product of the inputs and the weights. Consider a neuron which has a "N" inputs ($i_1, i_2,...i_N$) and each of the inputs have an associated weight ($w_1, w_2,...w_N$). The weight sum product of the inputs would be as follows: $(i_1 * w_1) + (i_2 * w_2) + ... + (i_N * w_N)$. The summation function can be more complex than that and sometimes also include an activation function. The activation function may include an algorithm to further process or process the output of the neuron differently.

3. Transfer function.

   The output of the summation function is transformed to a working output through an algorithmic process known as the transfer function. The simplest form of the transfer function would be to compare the output to a threshold and fire the output if the computed value exceeds the threshold value. The transfer is usually not linear and may follow a "S" curve shape when plotted and is known as a sigmoid function. Other functions may also be used.

4. Scaling and limiting.

   Scaling is the process of multiplying the value of the transfer function by a scale factor and adding an offset value. Limiting is the process of ensuring that the output does not exceed an upper limit or fall below a lower limit. Limiting would restrict the output value within a range independently of whether a similar step was taken in the previous transfer function.

5. Output function (Competition)

   In a simple topology, the output of the transfer function is the output value of the output function. However, in some topologies the developer may choose to modify the network so that the output of a neuron is compared to that of its neighbours. The competition determines if a neuron is active or not and also which neuron would participate in the learning or training process.

6. Error Function (Back Propagation)

   During the training process of a neural network, the error between the output value and the desired value is calculated. The error value may be used as it is, or squared, or cubed or transformed by some other mathematical function. The computed error value is then propagated to the neurons in the previous layers and the weighted connections of the neurons are adjusted in an aim of having the output of the neurons, which are closer to the desired value.

7. Learning function.

   The learning function modifies the weights of the connections for a neuron based on some algorithmic function. This is done so that the output for each neuron helps to achieve a value which is desired for each training iteration (epoch).

So far, an individual neuron has been defined. In order to form a neural network, the neurons have to be arranged in a specific topology and may constitute of several layers. There is an input layer, an output layer and there may be several intermediate layers. Figure 4.13 shows a multi-layer neural network with an input layer, an intermediate layer and an output layer.

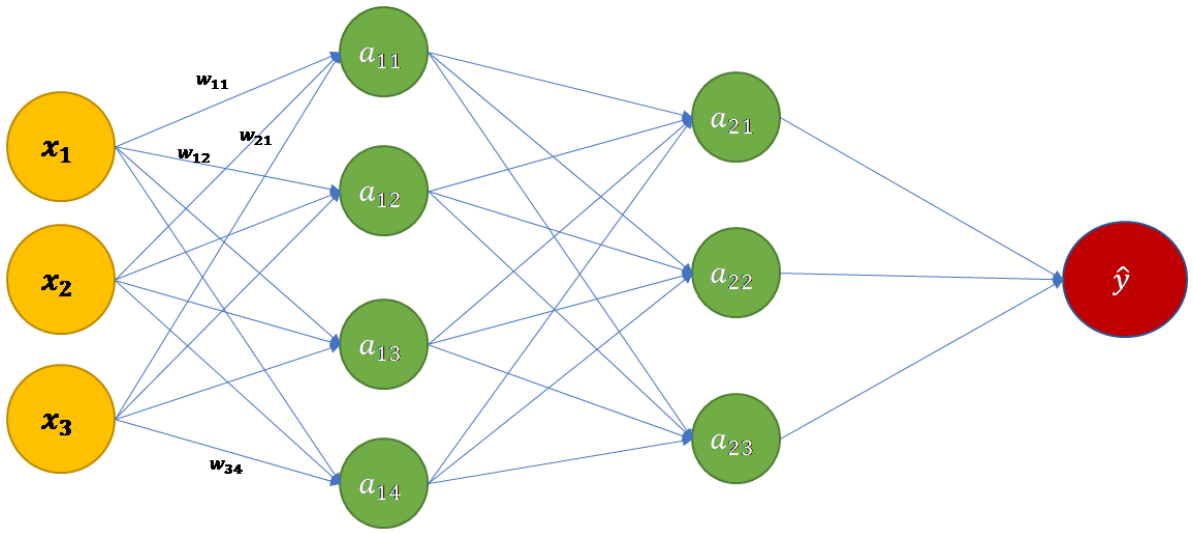


Figure 4.13: Representation of a neural network [170]

As shown above, the neurons from the input layer accept the inputs from the environment and the processing from the neural network can begin. As shown above each input neuron is connected to each neuron to the hidden layer. As outlined previously, the connections between the neurons are weighted connections and this weight and the input value are used to process the output of a neuron.

4.5.2.1 Training and calibration

Before a neural network can be used for any task, it has to be trained. During the training process, the neural network is exposed to a series of input values and their corresponding output values. During the training process, a series of parameters are used to adjust the weighted connections between neurons over several iterations, so that the output of the neural network gets closer to the expected output. The training may take a long time, sometimes even several hours, and it may be stopped by specifying a definite number of iterations or if the error (root mean squared value of the errors) fall below a certain threshold value. It may happen that a neural network is unable to reduce the error below the value of the root mean squared for a given training set or that the neural network cannot be trained sufficiently well to achieve the task. There are two main types of training and they are supervised training and unsupervised training.

In a supervised training set up, both the input and the output of the neural network are provided. The input of the network is processed through the network and an output is generated. The output is then compared to the expected output and the difference between the two, the error, is back

propagated in the network so that each neuron can adjust its weight in order to yield a result which is as close as possible to the expected output. The weights are continuously tweaked to improve the performance of the network. The data set used to train the network is known as a training set. Using the data set to train the network is known as an epoch and there may be many epochs used to train a network.

It may happen that a network never learns. This may happen if the training set does not have enough specific data from which the desired output may be derived. During a training cycle the user may define a certain number of epochs or if the root mean squared value of the error falls below a certain value. So even if the number of epochs is specified, the training may not be complete or conclusive in those iterations. If a threshold for a root mean squared value for the error is also specified, then the network may never achieve that value. Even if a network is trained, the user must ensure that the network can perform well for varied scenarios. The training set must not be too specific as the network would be good at solving a specific problem but may perform poorly to problems which were sparsely covered in the training set. It may be that a network simply cannot solve a problem. In that case the designer needs to review the inputs, the number of layers, the topology, the transfer functions, the activation functions and some more parameters to design and improve the network so that it stands a chance at solving the problem it is designed to.

The other type of training is called unsupervised learning. In that type of training of the artificial neural network, the inputs are provided but the output is not provided. The system itself decides how to organise the data and adjust the neural network. In this case most of the time the neural network detects trends and tendencies and then organize itself to process these trends. The way the network adjusts the weights and itself during the training is known as self-organisation or adaptation. Without an expected output, the network must be informed how to organize itself and this is usually done by the topology and learning rules. There is still a lot of research to improve the way unsupervised learning are set up and improve the output results [144].

For the network to adjust the weights, a learning rate is used. The learning rate is usually positive number between 0 and 1. The values in this range enable the network to train at a decent pace while also capturing the subtleties which may exist in the training set. In addition to a learning rate, a learning algorithm is also used. There are various learning algorithms but the common ones include Hebb's rule (Hebbian algorithm), Hopfield law, the Delta Rule, the Gradient Descent Rule

and Kohonen's law [145]. The neural networks can be created in different ways and the topology, the number of layers and the algorithms and parameters used determine the output of the neural network.

The research on neural networks is quite significant and it continues to be a preferred topic for research. In recent years, deep learning algorithms have proved to improve the accuracy of results in various sorts of applications of neural networks [146]. Developers and researchers have thus used deep learning algorithms on a series of applications including image recognition, prediction of time series (prediction of the stock markets) and natural language processing [147][148][149]. The computing power now allows developers to have several layers of neurons as part of the hidden layers of a neural network. The training time is now manageable and these "deep neural networks" can now be trained and used, thanks to the increase in computing power. They are preferred as they tend to get more accurate results as the increased number of layers take longer to train but are more reliable over time.

For the framework proposed in this thesis, a neural network was added as a mechanism to give the framework a learning ability. The potential and preferably ideal design for a domain would be stored in a database and seeded with a series of values. The neural network would then be used to predict a design based on these values. The design computed by the neural network would be an output from the framework. The second use of the neural network would be to compute the abstraction of the design from a thick grain architecture, to a coarse grain architecture and a fine grain architecture. The next section describes in detail how the neural network is applied to the framework.

4.5.2.2 Application of neural network to the framework

The aim is to use the neural network for two purposes: the first one is to predict design artefacts for a problem domain given seeded values representing an ideal design and secondly using the neural network to abstract the design to various levels of granularity. The neural network should be able to predict a design or rather suggest a design, given the past accumulated data or data known to the neural network. For this matter, it was decided to have a neural network which would be able to predict the elements of the design. However, since the elements of an architectural design are hard to predict on their own with a neural network, it was decided to use time series prediction and adapt it so that the design of a piece of software for a specific domain could be predicted.

In order to use time series prediction to represent elements of the design, the design has to be broken down into elements which could be represented with a series of numbers. It was decided to use the database for the solution and extend to hold the data to represent the design elements. The design elements are to be represented in two phases. Firstly, there would be parent elements and secondly children elements. Each child element must have a parent element whereas a parent element may or may not have children elements, although ideally a parent element should have children elements. The parent elements would be either use cases or class diagrams, and over time could be extended to represent various other kinds of design artefacts. For a use case, the parent element would be the use case with a name and the children elements would be the actor and their actions. Therefore, the database is used to define an ideal design with each element having a seeded value. The seeded value is expected to vary from 0 to 1 and the higher the value the more prominent the design element. Using time series prediction, neural network would then compute a new value for each element. The newly computed value could then be compared to a threshold value and accepted or rejected. The threshold could also be ignored and all design elements could be presented to the user, with the associated value.

Time series predictions deal mostly with a series of numbers and the goal of the neural network is to predict the next value. Usually, time series prediction is applied to predict stock market prices or commodity prices. The theory is developed around taking a series of numbers and predicting the next value for the near future. In this case the framework needs to predict a term which would be a word(s). The word or words would have to correspond to a design artefact or an element of a design artefact. Time series prediction is not inherently geared for prediction of words. Therefore, it is suggested that design elements be placed in the database. For each key design elements, there should be a series of values which are to be used for the prediction of the design elements. It might be best to seek the help of a domain expert for that task. The framework would then be able to predict a value for each design element and then propose a design.

The second use of the neural network is to assist in the abstraction of the design and it is further elaborated in the next section. The framework aims to also propose an abstracted design which would be categorized into three groups: a thick grain design, a coarse grain design and a fine grain design. The input from the requirements text would be closer to the fine grain architecture. The aim is to propose abstracted design which are each refined to show a higher level of the design

where the features and user stories are grouped into design elements which are less detailed. The goal is to achieve this by using seeded values from the database and again use time series prediction to propose a design. Thus, the neural network would also assist in abstracting the designing which is better suited for distributed systems.

4.5.2.3 Design abstraction

When the design is produced from the text, it can be considered as a design, which is closest to the fine-grained architecture. The design that is produced is directly obtained from the requirements and the requirements define the needs of the system. Therefore, the design obtained contains a lot of details and is not abstracted into an architecture. The level of details would depend on the work and style used by the business analyst, the usual agreed-upon levels of details needed by the architect and the task at hand. For example, if the business analyst creating a set of improvements for an existing system, it is expected that the level of details would be ample. For the sake of this research, the focus is the derivation of the design and therefore it is expected that requirement document would contain information which are detailed enough to elaborate a design. It is also expected that the design would be granular enough to obtain a design which is "low-level".

The term "low-level" can be vague and needs some explanation. In this context it is expected that the requirements would enumerate all the key actors interacting with the system. Once this is known, the possible or intended actions of the actor can also be derived. In case additional data is present in the ontology, that is added and the output of the neural network is also brought into contribution. Once all the design artefacts are produced, the design obtained can be considered as a fine-grain architecture. This architecture can be regarded as the finest grained architecture that can be obtained directly through the automation process. Given the requirements are processed automatically and enhanced by the neural network, the primary design which can be obtained from the text is considered as the starting block. The starting point for the elaboration of a design or an architecture is the requirements, even if it is performed manually. Since the design is automated and then enhanced, the design produced could not be more granular and therefore is considered as a “low-level” design, which is also referred to as the finest grain architecture which can be obtained.

However, the design obtained at this stage cannot be considered as the architecture of the whole system, and may not be adapted for a distributed system. Therefore, the framework will then

abstract the current design a less granular design which at this stage is referred to as the "coarse grained" architecture. This coarse grained architecture is achieved by grouping elements of the design which are to be coded and deployed on the same component. For example, all the methods for validating a user such as “insert card”, “enter pin”, “manage pin / card”, “verify user” can be grouped as the “Authentication”.

The abstraction of the design is made possible with the use of the neural network. The same strategy as before is intended and the user or experts would be solicited to provide a possible combination of abstracted design. The design is replicated in the database with a series of numerical values for each design element and level of abstraction. Then the neural network can compute a new predicted value for each element and decide to include that element in the output after comparing it to threshold. The strategy is the same but the way the data is represented and the way the output is created might differ.

The neural network would abstract the architecture into a coarse grain architecture and then into a thick grain architecture so that the components obtained from the previous step can be grouped into tiers / layers so as to obtain a high-level architecture. A multi-tier architecture, for example, a three-tier architecture, has a presentation tier (where the User interface is hosted), a business layer where all the application logic is computed and a data layer where all the backend operations and databases are stored. Thus, the components from the previous steps are abstracted to obtain the required number of tiers / layers so that an architecture can be derived.

4.6 User validation

The last stage is for the user or an expert to validate or modify the final results. The user is solicited earlier to either seed or provide some backend values to help set up the domain. The database must be set up for a domain and then for that domain the values for the ontology and the neural network would have to be seeded. That part of the user contribution could be considered as the configuration part where the user is setting up the framework with data so that the data dictionary, tables in the database, the ontology and the neural network would function, given a set of inputs. The configuration is setting up each domain so that the user may use the framework and is expected to be done prior to utilization. This is expected to be done by the developer or the user or an expert who is well versed in that domain. The data would have to be changed by the developer in order

to fit the table structures created and optionally the user may be presented with an XML file to configure the framework before using it for the first time. Thus, the user would be able to include additional domains to be used. However, the input from the user may not be entirely focused on the configuration but could also be solicited for correction and update of the data presented to him / her by the framework.

The user first uploads the file to be processed to the framework, and then waits for the latter to generate the UML artefacts. When the UML artefacts are generated, they are presented to the user. There are various ways in which the user may be solicited with an input which affects the output of the framework. For an artefact such as the class diagram, the framework makes use of the ontology to enhance the class diagram after it has been processed by the NLP parser. When the ontology is brought into contribution, a user defined weight component is used to determine if any item in the ontology is to be suggested and added to the design. There are several parameters involved and it is important to note that the output of the ontology may depend entirely on the user defined values. Therefore, the first contribution of the user could be to define the values in the ontology (user defined weights) so that the ontology can be used to enhance the output of the parser.

Once all class diagrams and use cases are ready, the framework then proceeds to abstract the architecture to a fine grain or coarse grain and thick grain one. The user could be asked to edit the final design, maybe in an XML file which would then be fed back into the database. In the final design the user would be able to validate the whole architecture, number of layers (and tiers if needed) to be used by the architecture. The architecture proposed is better adapted for a distributed system as it is abstracted and the user can really validate if the architecture proposed satisfies the need of the requirements. Coming down a level, the user can check if the “coarse grain” architecture is abstract enough (not too detailed) and logically groups the components which are to be used. This step may also help with the technical architecture when the application is to be coded. Finally, the user can validate the low-level architecture and to ensure that the application is correctly designed in order to deliver on the requirements of the project. At this stage, there is no plan to feed the output from the framework back into the database to create a self-learning system.

Once the user is satisfied with all the elements of the design, he / she could proceed to generate the artefacts for the design. The user input is used in the ontology to facilitate further processing

and enhancements on the following runs for a domain. The changes made by the user to other aspects of the design could also be captured by the learning system at a later stage. The neural network is also used on top of the ontology to predict future values and propose potential design elements and an abstracted design. The user validation could be an important phase as it allows the user to generate the artefacts, design and architecture that he /she really deems necessary for a given set of requirements. The partial automation is meant to help and accelerate the process of design and architecture.

4.7 Conclusion

Automation has been applied to various aspects of the SDLC which can accelerate key processes while reducing the risk of errors. In the previous chapters, a review of how automation has been introduced in some of the tasks of the SDLC and how it can impact a project. Since defects injected in the earlier phases of the SDLC can be costliest to fix, the frameworks aiming to reduce defect injection and accelerate the design process were also reviewed. The previous chapter analysed the existing approaches and demonstrated their capabilities and limitations.

The need for a new framework was explained and, in this chapter, the proposed framework was detailed and the main components outlined. The file uploaded by the user is first prepared, cleaned and parsed through the NLP component. The natural language processing component scans the raw input and then starts extracting concepts from the text. These concepts are then used to produce a use case diagram and a class diagram. The output is then enhanced with the help of an ontology. The ontology is maintained with user data or system (encountered by the system while processing input files) and is configured with a few parameters. Then the neural network analyses the data present in the database and also predicts a design. The time series prediction capability of the neural network is used to propose potential terms, actors, attributes and methods to the user. The design, regarded as the low-level design is then abstracted to a coarse grained architecture and a thick grain architecture, with the help of the neural network. In so doing, the framework is able to produce a design which is better suited for distributed system.

CHAPTER 5: FRAMEWORK DESCRIPTION & IMPLEMENTATION

5.1 Introduction

The task of developing a complex software system, can be a daunting one, filled with pitfalls. It is important to make the choice of technologies and libraries which are appropriate for the project. The literature review chapters have investigated some of the attempts to minimize the introduction of defects in the SDLC, including commercial tools, semi-automated tools and automated tools to assist in the elaboration of a design. After a review and critical analysis of the existing approaches, the need for a framework was explained. The previous chapter provided the details of the framework at a high level, with detailed discussion of the function of each of the main components. This chapter provides an explanation of how the framework is to be implemented and the potential technology that can be used.

The main components and their interactions were elaborated in the previous chapter. The first step starts when the user selects and loads a file to be processed to the framework. The first component to interact with the user input is the NLP component which cleans the data and extracts meaningful concepts from them. There are several lists which stores different kinds of concepts which can then be used later. The design extraction component uses the data from the NLP component and builds the design artefacts in turn. The ontology is also solicited in assisting with the design creation process. After the use cases and the class diagrams are created, the next step is to create an architecture by abstracting the design. The design is abstracted using a custom algorithm and a neural network to produce an architecture. After the architecture is produced, the user is presented with the results and can validate the design and the architecture. This chapter provides a walkthrough of the different steps and they are meant to be executed and implemented at a technical level.

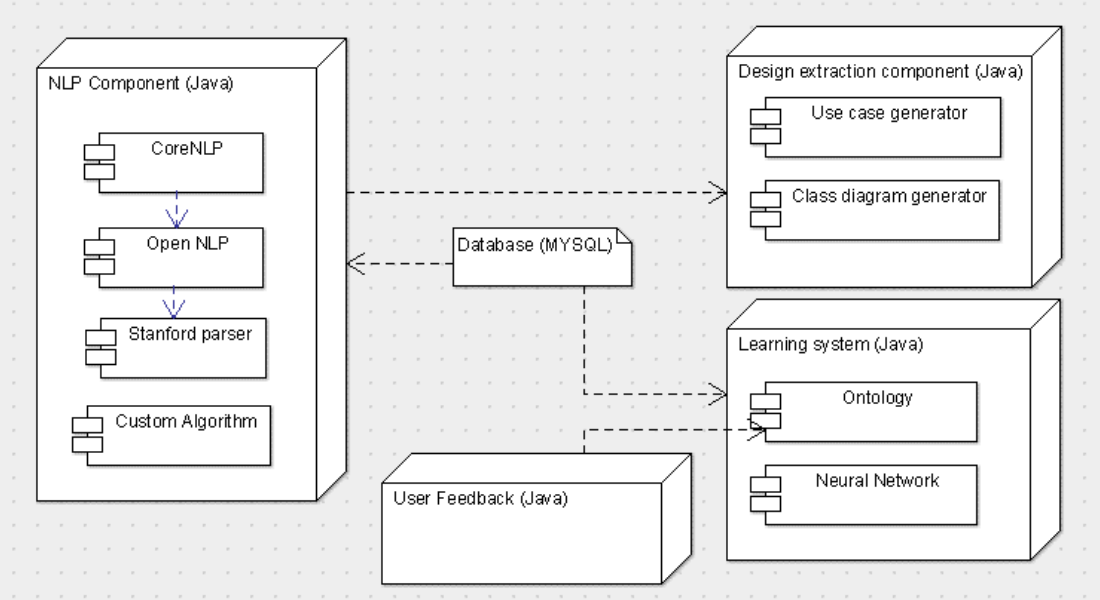


Figure 5.1: Deployment diagram for the framework

Figure 5.1 shows a deployment diagram of how the different components interact with each other, the technology stack used for each component and how they all build up the solution. The rest of this chapter provides a more detailed description of each of the individual component.

5.2 NLP component and parsers

The aim of the NLP component is to process the raw data so that meaningful concepts can be extracted from the text, for future modules to be able to elaborate a design from these concepts. The NLP component is used to clean the data, remove unnecessary values from the text and refine the input. The text is then parsed and the processing rules are applied to extract meaningful concepts from the text. These concepts would then be further used by the next component, the design extraction component so that a design can be put together. The idea is to use the NLP component to process the raw data in several phases so that meaningful concepts can be extracted and used for the derivation of a design. The processing rules are inspired from the literature and the aim is to eventually extract a design, regardless of context and the domain. Therefore, the point of the NLP component is not to understand the text, but to extract concepts from it.

5.2.1 Technologies and implementation

There are several processing stages within the NLP component and the output from each stage of processing is used by the next (inspired from slicing). Program slicing is the computation of the

set of program statements (the slice), that, at some point of interest, affects the values of the program. In our case, slicing would mean having a holistic look of the desired output and determine each key step (slice) that would help to transform the input into the output. If the whole program is viewed as an onion, then each round of processing can be viewed as one layer of programming which helps transform the data from the current input to get to the next. In order to create the NLP component, a series of external libraries, custom algorithms, data structures and a database were used. This section provides a walkthrough of each of the step in the NLP component and the associated technologies, libraries or algorithms that were used.

As explained in the previous chapter, there are six main steps in the NLP component which are to parse, extract concepts and produce data structures. These steps are:

1. Eliminating special characters.
2. Coreference resolution.
3. Isolating sentences.
4. Reduction of concepts.
5. NLP processing rules.
6. Concepts identification and elaboration.

Each of the above-mentioned steps in the NLP component are elaborated in the next sections, mentioning the technology that was used to implement each step.

5.2.1.1 Eliminating special characters

The first step is to eliminate special characters and this was achieved using a custom logic, without the use of external libraries. The special characters which are not required in the text can be replaced with another. This is achieved by using a table in the database where all the special characters and the replacement value are listed in the database. Most of the undesirable characters are simply replaced by a space or a blank character and this approach allows the user to extend the list of characters to be replaced by adding values in the database without changing the code.

5.2.1.2 Coreference resolution

The second step is coreference resolution and it aims to replace pronouns (or other words) like “he”/ “she” by the word that it is actually pointing to. During the initial phases of development, it was noticed that a corpus could contain pronouns, such as the word “it”, which are scattered across the text and each instance could point to a different meaningful word. The part of natural language processing resolving these pronouns is called coreference resolution and it was decided to implement this facility in the framework. This is done to avoid design elements and classes using word like “it” for a class or an actor. This is achieved by using the CoreNLP java libraries from Stanford [150]. The Core NLP libraries is a series of Java libraries made available to the public and the libraries have three dedicated purposes, which are:

- Named Entity Recognition
- Coreference Resolution
- Identifying basic dependencies

The libraries were used exclusively for coreference resolution and the facilities of named entity recognition and identifying basic dependencies are not used as they are not needed. The version of the library used is 3.6.0 and the four java archive files (jar) files used are:

stanford-corenlp-3.6.0.jar

stanford-corenlp-3.6.0-javadoc.jar

stanford-corenlp-3.6.0-models.jar

stanford-corenlp-3.6.0-sources.jar

The Core NLP library was chosen for a number of reasons. Firstly, the libraries are created and updated by the Stanford Natural Language Processing Group, which is a unit of research within the Stanford University and the group is dedicated to research on natural languages. The Stanford Natural Language Processing Group is a well-established and reputable unit and it is dedicated to research on natural language processing and have been researching the topic for more than 15 years. Updates to the libraries are made available to the public when newer version of the libraries are released. The releases typically include bug fixes and / or additional features, data structures

or methods which are added to enhance the capacity of the Core NLP libraries. The library is coded in Java and can be integrated with almost any existing Java program, provided the libraries are downloaded, saved and correctly referenced within the project. It was planned to code the framework in Java and the Stanford Core NLP libraries are readily available in Java and can be used as off-the-shelf plugins. It is to be noted that the Core NLP library is not a parser and provides only the three above-mentioned features which can be used to assist in NLP tasks.

5.2.1.3 Isolating sentences

The third step is to isolate sentences. It was noticed that sentences were occasionally not correctly identified as a unit, when using the Stanford parser. Or sometimes when an excessively long sentence is created by using the word “and”, parsers identify multiple subjects, verbs and objects. This makes it harder to identify concepts which are later used for the elaboration of a design. It was therefore deemed necessary to separate or isolate sentences so that complete individual sentences were processed as a unit rather than multiple sentences being amalgamated into one and being processed. In order to achieve this the OPEN NLP parser was used [151]. OPEN NLP is an open-source NLP library, operating under the aegis of the Apache Software Foundation and the library is also coded in java. Similar to the Stanford parser, there are regular versions made available to the public. As it is also coded in Java, it can be used in the same Java program by invoking the **J**ava **Ar**chive (jar) files as a reference to that project. The OPEN NLP parser is a much more complete natural language processing library and has many capabilities including a parsing capacity and coreference resolution. However, within the framework, OPEN NLP is used only to isolate sentences and not for anything else. When OPEN NLP was used for parsing, it was noticed that it had quite low accuracy. For example, nouns were not always correctly identified as well as the subjects and objects in a sentence. Therefore, OPEN NLP was not chosen as the parser but was used for its off-the-shelf feature to isolate sentences as it does it quite well. OPEN NLP is used for this feature and uniquely for this feature and the libraries used are:

opennlp-tools-1.6.0.jar

opennlp-tools-1.6.0-sources.jar

5.2.1.4 Reduction of concepts

The fourth step is the reduction of concepts and it is also achieved in a similar way as the replacement of special characters. There is a table in the database which stores all the synonyms which are to be replaced throughout the text. This avoids having many similar words and concepts pointing to the same entity. This approach also allows the user to replace symbols like "€" with the word "Euro" if that would be more convenient or help maintain consistency. Therefore, the design can be fluid with an entity represented only once and not multiple times with very similar, comparable and interchangeable words.

5.2.1.5 NLP processing rules

The fifth step consists of processing the text through the rules so that they can be further streamlined and prepared for the design derivation. The rules were mentioned in the previous chapter and they are repeated here for the sake of consistency:

1. Identify the last word of each sentence. Save these words in the list Stop words.
2. Keep track of the total number of words in the text. Also keep a running count of the number of occurrences of each word.
3. Calculate the frequency of each word.
4. Use an NLP (Stanford) parser to parse the whole document.
5. Store all the nouns in a list, the Nouns list.
6. Store all the verbs in another list.
7. All the nouns are saved in the concepts list.
8. All the subjects of the verbs (regardless of whether they are nouns) are also added to the concepts list.

There are eight processing rules which transform the data from the NLP component into key concepts which are then later used for the elaboration of a design. Firstly, all the last words of each sentence are saved in an array list of type "string" and are saved later for future processing. The second step is to identify the running count of each word and the frequency of each word. This is achieved by saving all the words for the current processing cycle in the database. After all the

words are saved, the total number of words is calculated and the number of times each word occurs, the frequency of each word is added in the database, which is the third step.

The fourth step, in the NLP processing rules, is to use a parser to parse the whole document and the Stanford parser was chosen. The parser is also made available by the Stanford Natural Language Processing Group and java archive files are downloaded and added to the project. The Stanford parser is backed by the research at Stanford University and the first library was made available to the public in 2002 and there have since been regular releases which improve the accuracy of the parser, contain added features and support additional languages. When using the Stanford parser, for the English language, there are different kinds of parsing algorithm which can be used. For the case of this framework, the English PCFG (probabilistic context free grammar) parsing algorithm was used [152]. The parser chosen must be loaded in the parser so that the parser would know which parsing algorithm to invoke. After the parsing is complete, the data structure returned is a parse tree, custom to the Stanford parser, which is similar in concept with a parse tree as it is understood in NLP. The documentation of the data structure is available on the Stanford parser website along with the publications (research papers published to back their research), and downloads for other languages including Arabic, Chinese, French, German and Spanish [153–157].

The algorithm within the framework loops through the parse tree provided by the parser and processes the sentences one at a time. For each sentence, the nouns (potential actors, classes, and attributes) are saved in a list and all the verbs (potential actions in use cases, and methods) are also saved in a different list. The framework is thus able to track all the subjects, verbs and objects for each sentence and all this data is saved for later processing. In order to derive more statistics from the text and each individual parse, all the data are also stored in the database. Every time, a file is processed a unique id is created to track all the data for that round of processing. Then all the words are stored in the backend, along with the part of speech tag (noun, verb, preposition, etc…). This data is later used by the processing rules to create the design artefacts and the architecture. The fifth step is covered as the list of nouns is created and the sixth, seventh and eighth step are also covered as the list of nouns, verbs and concepts are now ready and this wraps up the NLP parser usage and the NLP processing rules.

5.2.1.6 Concepts identification and elaboration

The final and sixth step in the NLP component is the identification and elaboration of concepts. Here again the rules were mentioned in the previous chapter and repeated here.

- All the concepts (coming from the noun list) which have a frequency of less than 2 % are ignored for future processing.
- All the subjects of the verbs are considered for future processing, regardless of frequency.
- Words which are too generic, or correspond to a design element, a proper name or a geographical location are ignored for future processing. The database stores a list of words which may be ignored as they are irrelevant or too generic. For example, words like “machine”, “function”, “London”, “John”, etc… can be added to that table so that they are ignored.

All of the transformation steps of the NLP component have been explained in details, with reference to the external libraries used or how either a custom logic or database were used to accomplish the transformation steps. The input to the NLP component is a text file, which describes the requirements. The six transformation steps mentioned above then clean, parse and extract meaningful concepts from the text. The output is a series of lists (nouns, concepts, stop words) and data structures (parse tree from Stanford parser or custom data structures) which are used for the next steps.

5.2.2 Justification of the chosen technologies

The framework was developed as a result of a project with the research group behind the ZDLC suite of products [158]. Since the tools in the ZDLC suite of products were coded in Java, it was decided to use the same programming language and technology stack to create the framework. During the literature review, it was also noticed that a few of the existing frameworks were coded in Java [87][88] and there are a few parsers, taggers and other NLP libraries readily available in Java [150][151]. It was therefore decided to have the programming and development environment as Java.

After devising the key steps for the processing rules for the NLP component, it had to be implemented. Given the wide range of support and online libraries available, it was decided that

there was no need to create a custom parser or toolset, but a custom algorithm to integrate these libraries to deliver on the framework's needs. For the first step of coreference resolution, after some research the CoreNLP library was found and it was used for that step. The reason behind this decision relies on the fact that the library is developed by the Stanford Natural Language Processing Group and it is supported by the research papers which have been published over the years. The research unit within the Stanford University is a trustworthy research group and have been carrying out research on NLP and NLP related research for over 15 years.

For the step of coreference resolution, the CORENLP library was selected and it is made available by the Stanford Natural Language Processing Group. The CORENLP provides coreference resolution as an off-the-shelf facility and does not require any extensive research to use the library. The reputation and long-standing track record of the Stanford University validates the choice of the library. For the same reason, the NLP parser chosen was the Stanford parser (a different library from CORENLP) to parse the whole document. During the literature review, it was also noticed that the UMGAR framework [88] also used the Stanford parser. For some of the next few processing steps, a custom logic was used with the help of the database. When there was a need to segregate sentences and isolate sentences, the OPEN NLP parser was used. OPEN NLP is a full-fledged NLP parser and can accomplish a lot of the NLP tasks, including parsing. During the initial phases of research, the OPEN NLP was used as a parser to parse the whole document. However, it was noticed that the accuracy was lower than the Stanford parser and could lead to incorrect identification of concepts. The nouns and the subject of the verbs were not always identified correctly. The OPEN NLP parser can be trained and it was trained on a corpus of 100 000 words and the accuracy of the parser was improved but not satisfactorily. It was, therefore, decided not to use the OPEN NLP parser as the main parser. At the same time, it was noticed that the OPEN NLP parser had some off-the-shelf capacity to isolate sentences and it was decided to use the OPEN NLP parser to isolate the sentences. Without this step, the Stanford parser can sometimes attempt to parse multiple sentences in one go and could return a parse tree which would contain the data for multiple sentences.

5.3 Design derivation

All the processing done so far have identified the concepts and shortlisted words which may be of interest. This data is either saved in the database, in lists (or arrays) and other data structures. The

next step is to use this data and create design artefacts. Before diving into the process of how these design artefacts are created, there is a short introduction on the design models, including the ones that the framework would be producing.

5.3.1 UML diagrams

Models help us understand a system by simplifying some of the details and also allows the architect to view the intricacies of the possible system. The choice of what to model, how to model and what is the scope of the model can have an enormous effect on the understanding of the problem and the proposed solution. Unified Modelling Language (UML) has been around since the late 80's and has formalized the process of modelling and creating design artefacts where people can have a common understanding of their meaning [159]. Over the years, UML has grown in popularity and was prominently used in modelling object-oriented applications, as the roots of UML can be found in the early days of object-oriented programming (OOP) [160]. Over the last decade, web applications have grown in popularity and UML models have also been used to model these applications. UML is therefore not confined to OOP but can encompass a larger trend to model applications.

There are two main categories of UML diagrams and they are behaviour diagrams and Structure diagrams, as shown in figure 5.2.

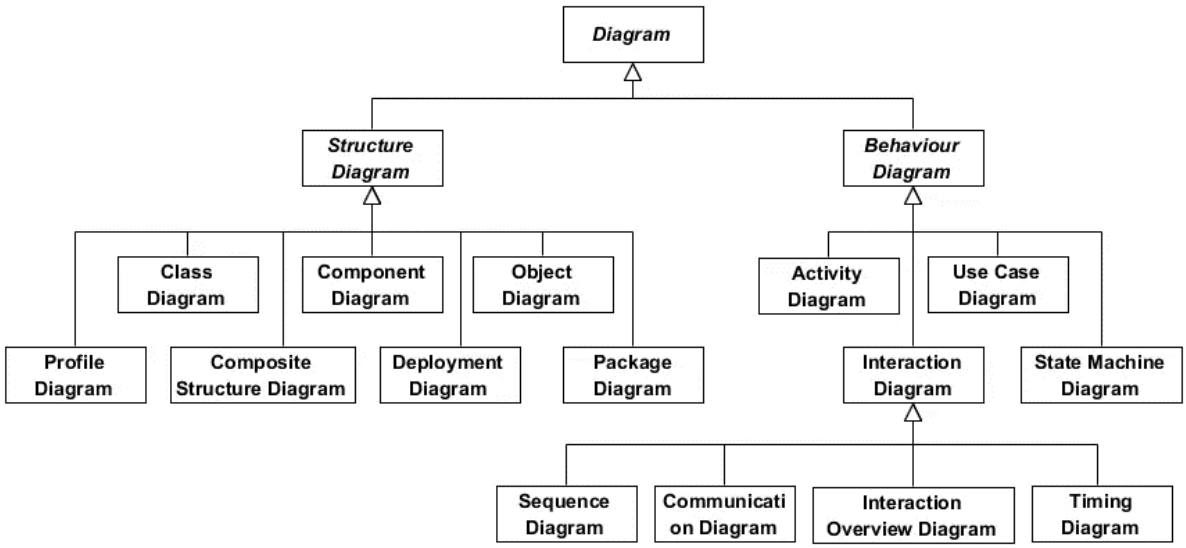


Figure 5.2: Types of UML diagrams [161]

There are different diagrams that can be created to describe how a software system can be developed. Amongst the behaviour diagrams, the use case diagram and the sequence diagram are very popular to describe the behaviour of a system, particularly object-oriented design, including the first generation of mobile coding (J2EE applications for mobiles). Amongst the structure diagrams, the class diagrams, component diagrams and deployment diagrams are quite popular.

5.3.1.1 Use case diagram

A use case diagram is a representation of a users' (or user groups') interaction with the software system which shows the relationship between the users and the different interactions with the system. A use case diagram gives a visual representation of all the different types of users (user groups) for the whole system and how each user group interacts with the system. Each user group is represented by a stick figure, known as an actor and each action is represented by an ellipse, known as a use case.

5.3.1.2 Sequence diagram

A sequence diagram shows object interactions arranged in time sequence. The object and class interactions are shown sequentially, in the order of time that they are scheduled to happen and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. A sequence diagram consists of parallel vertical lines (*lifelines*), different processes or objects that live simultaneously and horizontal arrows, the messages exchanged between them, in the order in which they are meant to occur.

5.3.1.3 Class diagram

A class diagram is a static structure diagram that describes the structure of a system by showing the system's classes, the attributes and methods of these classes and the relationships among the objects. The class diagram is the main building block of object-oriented modelling and it is used for general conceptual modelling of the structure of the application. Class diagrams can be easily converted into code and also be used for data modelling. The classes in a class diagram represent both the main elements, interactions in the application, and the classes to be programmed.

5.3.1.4 Component diagram

A component diagram depicts how components are connected together to form larger components or software systems. Component diagrams can provide an easy way to illustrate the structure of

complex systems. Component diagrams are also used as a communication tool between the development team and other stakeholders of the system. Programmers and developers use the diagrams to formalize a roadmap for the implementation, allowing for better decision-making about task assignment or skill improvements required.

5.3.1.5 Deployment diagram

A deployment diagram models the physical deployment of artefacts on nodes. For example, if a web site is to be described, a deployment diagram would show what hardware components ("nodes") exist (a web server, an application server and/or a database server), what software components run on each node (web application, database), and how the different pieces are connected (JDBC, REST). The nodes shown appear as boxes and the artefacts allocated to each node are illustrated as rectangles within the boxes. Nodes may have sub-nodes which are depicted as nested boxes.

5.3.2 Process implementation

For the purpose of this framework, it was decided to have use cases and class diagrams as design artefacts. The use cases can depict the interactions of the key stakeholders with the system which is convenient to show all the key roles and actions intended. The class diagram is an important part as it can be readily used for OOP programming. From these design artefacts, an abstracted design is to be created to make the architecture more suited for modern architectures like distributed systems.

5.3.2.1 Deriving use cases

Firstly, it was decided to make the framework attempt to derive use cases. Use cases were typically used to describe object-oriented designs and it is very close to the modern practice of user stories to produce a design. Use cases are a well-established UML design artefact and can give the high-level picture of what the software needs to do. The advantage of having use cases is that it is not bound by any technology, even though it is usually associated with object-oriented programming and can be easily translated into user stories which are used to derive a design for distributed systems such as service-oriented architecture or micro-services.

Cockburn claims that a use case is a collection of possible sequences of interactions between the system under discussion and its external actors, related to a particular goal and an action connects

one actor's goal with another's responsibility [162]. The actors can be people (or user groups) as well as other computer systems. The overall use case diagram should demonstrate the interaction of the key stakeholders with the system and the main interactions (actions) that they should be able to execute. When all the use cases are depicted in a use case diagram, it gives a high-level picture of what the software system would look like, all the key stakeholders and all the main interactions or actions intended for that software system.

The NLP component was explained in details previously and from the output of the NLP component, each sentence is isolated and for each sentence there is a parse tree (obtained from the Stanford parser). Using that information, for each sentence, the verbs are known and the subject and the object are also known. For example, the sentence "`At the core of banking activities is its transaction processing system.`" would be tagged as follows "`At-IN the-DT core-NN of-IN banking-NN activities-NNS is-VBZ its-PRP$ transaction-NN processing-VBG system-NN .-.`"

So, the word "the" is tagged as a determinant and the word "core" is tagged as a singular noun. In order to extract more information from these words, it would be necessary to get information from the relationships between these words. The data from the parse tree can be extracted and it would look like the following structure.

```
case(core-3, At-1)
det(core-3, the-2)
nmod:at(system-11, core-3)
case(banking-5, of-4)
nmod:of(core-3, banking-5)
nsubj(system-11, activities-6)
cop(system-11, is-7)
nmod:poss(system-11, its-8)
compound(system-11, transaction-9)
amod(system-11, processing-10)
root(ROOT-0, system-11)
```

The second line can be read as the relationship for a "determinant" and the word "the", (followed by the number two denotes that it is the second word) and it is the determinant to the third word

“core”. By going through information like that for each sentence, the framework uses a custom algorithm to determine all subjects, verbs and objects for each sentence. The list of relationships between each pair of words is looped through and all the verbs, the subjects of these verbs and the direct objects of these words are identified and retained for further processing. In order to proceed to the next step and derive use cases, the following rules are then applied to construct the use case:

- If the subject of the verb, is a valid concept (passes all the validation rules), then that concept is an actor.
- Each sentence is processed again and if there is a valid actor in that sentence, the tense of the verb is checked. Present tense is the expected tense and other verb tenses such as the past tense or the past participle are ignored.
- The sentence is then split on the verb and the remaining part of the sentence (after the verb) is treated as the action part of the use case. For this reason, it is important to isolate sentences and avoid extremely lengthy or long sentences.
- The list of use cases obtained so far needs to be refined, using the following rules:
    - Once again, the algorithm loops on all of the sentences in the corpus.
    - When treating the action part of a sentence, the algorithm checks if there are no other valid use cases in that part of the sentence. If there is one, then the use case is split into two use cases each describing a specific action.
    - Duplicate or similar use cases derived from the same sentence are eliminated.
    - The verb and its direct object are retained for further processing. It would be used for the elaboration of a class diagram.

After the above processing rules are applied on the whole of the corpus, there is now a list of use cases which are deemed valid for the corpus. They are stored in a custom data structure which is a list of “use cases” where “use cases” are a custom defined class which is created to store all the data about a potential use case. A first list of use cases is created by the framework based on the rules and processing logic described above. That list is then filtered so that duplicate or use cases

appearing multiple times are removed from the list. The list of use cases is now complete and the next step is to start deriving the class diagrams.

5.3.2.2 Deriving class diagrams

UML class diagram allow for modelling the static structure of an application domain, in terms of concepts and relations between them [163]. Class diagrams are very popular to describe object-oriented software architecture and many of the key features of object-oriented architecture (like inheritance) are directly relatable to class diagrams. The practice of coding classic object-oriented applications is done by creating classes, with attributes and methods. Integrated development environments (IDE's) such as Microsoft's Visual Studio and Eclipse IDE provide a code generation facility and can generate all the codes for a class (class definitions, interface, constructor, methods and attributes) provided that the user has used the graphical user interface to draw a class diagram, without errors.

It was decided that the next design artefact to be produced by the framework would be a class diagram. The elaboration of class diagrams uses the output from the NLP component and builds on the work to create use cases to derive class diagrams. The following steps are taken to build the class diagram. All the actors retained from the use cases are considered as classes. The algorithm loops through the parse tree once more and identifies all the verbs and direct object for that verb and combines the verb and the object to create a method for that class. The list of methods is compiled for each distinct class. The text is scanned once again and tries to identify two consecutive nouns in the text. When two consecutive nouns are found and the first noun is a class, the second noun is retained as an attribute. In so doing a list of classes with their attributes and methods are created. The class diagram is enhanced with the ontology and that is described in details in the next chapter.

5.3.3 XML

The preferred output of the framework is to create an XML file which represents the design artefacts as the output of the framework. XML has been used since the early 2000's to represent UML artefacts [164][165]. XML is a handy way of representing UML and over data structures as it is quite versatile and can be used to represent hierarchical data, linear data or a combination of both and can also easily store a large record set with a lot of individual data records. There is a

standard to describe UML diagrams and it is the XMI file, which stands for XML Metadata Interchange [166]. The XMI standard helps to maintain an agreed standard so that UML diagrams can be expressed as XML and be readily understood and interpreted by all those adhering to the standard.

For the purpose of this framework, it was also decided that the output would be an XML file. The XML file is chosen as it can express the design artefacts, comprising of the class diagrams and the use case, the component diagram and the abstracted architecture. The output XML file would be a custom one as it would store all the outputs from the framework.

5.4 Framework storage

In order to store all the runs and respective processing of each round of processing, it was decided to use a database. The choice was a classic database rather than a cloud hosted, light weight database as there is then no need for an internet connection to run the program and there is no need to buy a license just to access the database. The data gathered by the database over time, can later be processed as part of a big data algorithm. It was decided that the database would be a MySQL database as it can be locally hosted, does not require any cost or licensing and can be run without an internet connection (when run locally on a machine). Furthermore, the database server can be hosted and accessed remotely, if there is a need for it, and there is plenty of documentation and troubleshooting assistance which are available online.

5.4.1 Process implementation

In order to set up the database for the framework, the tables were created and this section provides a walkthrough of the main tables needed and used by the framework. Firstly, the domain table must be set so that all the different domains for which the framework is intended can be listed. For example, data for the ontology and the neural network would be stored in the database based on domains. After all the domains are listed, the “run” table must be set up. The run table is a table which generates a unique id each time the software is run. The data for each individual time the software is run would be tied to a “run id” and in conjunction with a “domain id”.

The above-mentioned tables are the generic ones which are created. The next set of tables are used by the NLP component. The word count (word_count) is then created so that the frequency and all the individual words in a corpus can be logged and counted. Every time a word is encountered in

the corpus it is added to the table or if it already exists, for that run, the word count is increased. After the file is processed, the frequency of each word is computed and all the data is grouped by the run_id and the table definition are shown below.

Field *	Type *	Null *	Key *	Default	Extra *
wrd_Id	int(11)	NO	PRI	{null}	auto_increment
wrd_Word	varchar(255)	YES		{null}	
wrd_Count	int(11)	YES		0	
wrd_Frequency	float(11,4)	YES		0.0000	
wrd_TotalCount	int(11)	YES		0	
wrd_run_Id	int(11)	YES	MUL	{null}	

The first three tables which are needed every time the program is executed are the "run", "word_count" and "concept_word" tables. The table "run" is used to generate a unique integer identifier every time a file is processed and this identifier is then used to track all the words corresponding to that particular run. Optionally a description can be included for that processing, if it may help the user to identify the processing of a particular file. Then the "word_count" table is used to keep track of each and every word in that file. After the processing rules are applied, certain words may be identified as concepts and would then be saved in the "concept_word" table. The relationship between the tables are shown below:

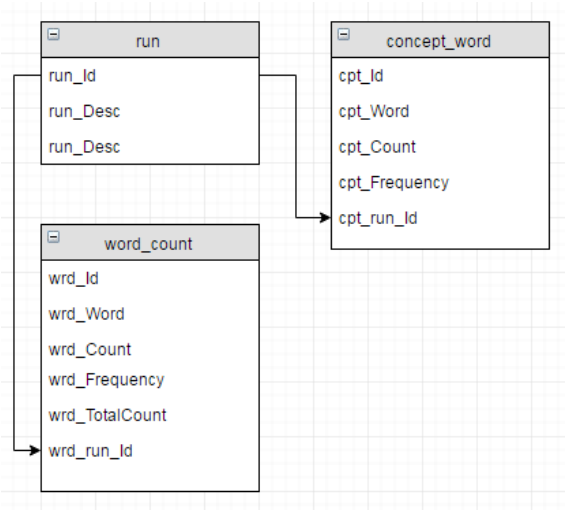


Figure 5.3: Run, Word Count and Concept Word tables

Other information for the NLP component, such as the nouns and the stop words are saved in lists. The processing rules from the NLP component then shortlists all the concepts that are deemed

valid. During the processing, special characters and synonyms (or similar concepts) are replaced. There are two tables which store the data for the special characters and the synonyms. There is also another table which lists all concepts which are to be excluded from the design. After all the special characters, synonyms and concepts from the exceptions list are removed, the remaining valid list of concepts are saved in the concept_word table. The four tables assisting the NLP component with the processing rules are listed below:

Special characters:

Field *	Type *	Null *	Key *	Default	Extra *
spc_Id	int(11)	NO	PRI	{null}	auto_increment
spc_character	varchar(20)	YES		{null}	
spc_replacement_text	varchar(20)	YES		{null}	

Synonyms:

Field *	Type *	Null *	Key *	Default	Extra *
syn_Id	int(11)	NO	PRI	{null}	auto_increment
syn_root_word	varchar(255)	YES		{null}	
syn_second_word	varchar(255)	YES		{null}	

Exception list:

Field *	Type *	Null *	Key *	Default	Extra *
exp_Id	int(11)	NO	PRI	{null}	auto_increment
exp_Word	varchar(255)	YES		{null}	
exp_Type	varchar(255)	YES		{null}	

Concept word:

Field *	Type *	Null *	Key *	Default	Extra *
cpt_Id	int(11)	NO	PRI	{null}	auto_increment
cpt_Word	varchar(255)	YES		{null}	
cpt_Count	int(11)	YES		0	
cpt_Frequency	float(11,4)	YES		0.0000	
cpt_run_Id	int(11)	YES	MUL	{null}	
cpt_weight	decimal(5,2)	NO		0.00	

These are the main tables in the database which are used for the NLP component. The next phase of processing is the design derivation component and it was mentioned earlier that the ontology would be solicited to complement the design extracted from the text. There should be pre-populated data in the backend for the ontology to work. There are three tables which are used to store the ontology for the class diagram and the tables are classes, class_attributes and class_methods. The "classes" table stores the ontology for the classes for that domain, with the user weight component and the system weight component. If a class is accepted by the algorithm (computed value exceeds the threshold value), the algorithm goes on to compute the values for the attributes and the methods for that class and then again compares each of the computed values to the threshold. Some of attributes or methods which are flagged as mandatory values do not have their computed weight checked against the threshold. The tables which store the ontology are listed below:

Classes:

Field *	Type *	Null *	Key *	Default	Extra *
cls_Id	int(11)	NO	PRI	{null}	auto_increment
cls_domain_id	int(11)	YES	MUL	{null}	
cls_class_name	varchar(255)	YES		{null}	
cls_user_weight	double(5,4)	NO		{null}	
cls_system_weight	double(5,4)	NO		{null}	

Class_attributes:

Field *	Type *	Null *	Key *	Default	Extra *
cla_Id	int(11)	NO	PRI	{null}	auto_increment
cla_cls_id	int(11)	YES	MUL	{null}	
cla_attribute_name	varchar(255)	YES		{null}	
cla_attribute_type	varchar(255)	YES		{null}	
cla_system_weight	double(5,4)	NO		{null}	
cla_user_weight	double(5,4)	NO		{null}	
cla_is_default	tinyint(1)	YES		0	

Class_methods:

Field *	Type *	Null *	Key *	Default	Extra *
dm_Id	int(11)	NO	PRI	{null}	auto_increment
dm_cls_id	int(11)	YES	MUL	{null}	
dm_method_name	varchar(255)	YES		{null}	
dm_system_weight	double(5,4)	NO		{null}	
dm_user_weight	double(5,4)	NO		{null}	
dm_is_default	tinyint(1)	YES		0	

After the design has been obtained from the design component, the next step is to abstract the design into an architecture. The use cases are used as the starting point for the abstraction of the design. The use cases are chosen for their descriptive nature and provide a good starting point for the abstraction into an architecture. The architecture is planned to be technology agnostic and is not linked to any technology, programming language or platform. The use cases are taken as the starting point as they would describe the key transactions of the software system, the key stake holders and the main interactions within the system. The information present can be used to design traditional object-oriented programming or can be converted into user stories to describe more modern distributed architectures. The architecture could also be used to devise the client-based architectures like single page applications, which tend to be serverless (but use web services to query and save data to a backend) and are coded mostly in flavours of JavaScript and web technologies. The point of devising the architecture at this stage is to describe an overall architecture, without the need to choose the implementation technology.

In order to get started with the architecture, the use cases were used as the starting point. The “bag of words” approach was used to group and collate words which tend to describe the same kinds of operations together so that use cases or the data from use cases can be grouped as a low-level transaction. After the low transactions are identified, a series of medium level transactions can be identified and from this set of medium level transactions, a set of high-level transactions can be obtained. In order to achieve all of this, four tables are used. The first table contains all the words which would constitute the “bag of words” which identify a low-level concept and the table’s name is bag_of_words_usecase_lowlevel. From the low-level transactions obtained, the next set of medium transactions are to be derived. The list of low-level transactions which could potentially be grouped as medium level transactions are stored in the table low_level_transactions. The medium level transactions which could be grouped as high-level transactions are stored in the table

medium_level_transactions and the high-level transactions which they correspond to, are stored in the table high_level_transactions. The four table definitions are listed below:

bag_of_words_usecase_lowlevel

Field *	Type *	Null *	Key *	Default	Extra *
bwu_Id	int(11)	NO	PRI	{null}	auto_increment
bwu_domain_id	int(11)	YES	MUL	{null}	
bwu_llt_id	int(11)	YES	MUL	{null}	
bwu_words	varchar(255)	YES		{null}	
bwu_verb	varchar(255)	YES		{null}	
bwu_noun	varchar(255)	YES		{null}	
bwu_word_count	int(11)	YES		{null}	

low_level_transactions

Field *	Type *	Null *	Key *	Default	Extra *
llt_Id	int(11)	NO	PRI	{null}	auto_increment
llt_mlt_Id	int(11)	YES	MUL	{null}	
llt_mlt_description	varchar(255)	YES		{null}	
llt_transaction	varchar(255)	YES		{null}	
llt_concept_wording	varchar(255)	YES		{null}	
llt_concept_description	varchar(255)	YES		{null}	
llt_domain_id	int(11)	YES	MUL	{null}	

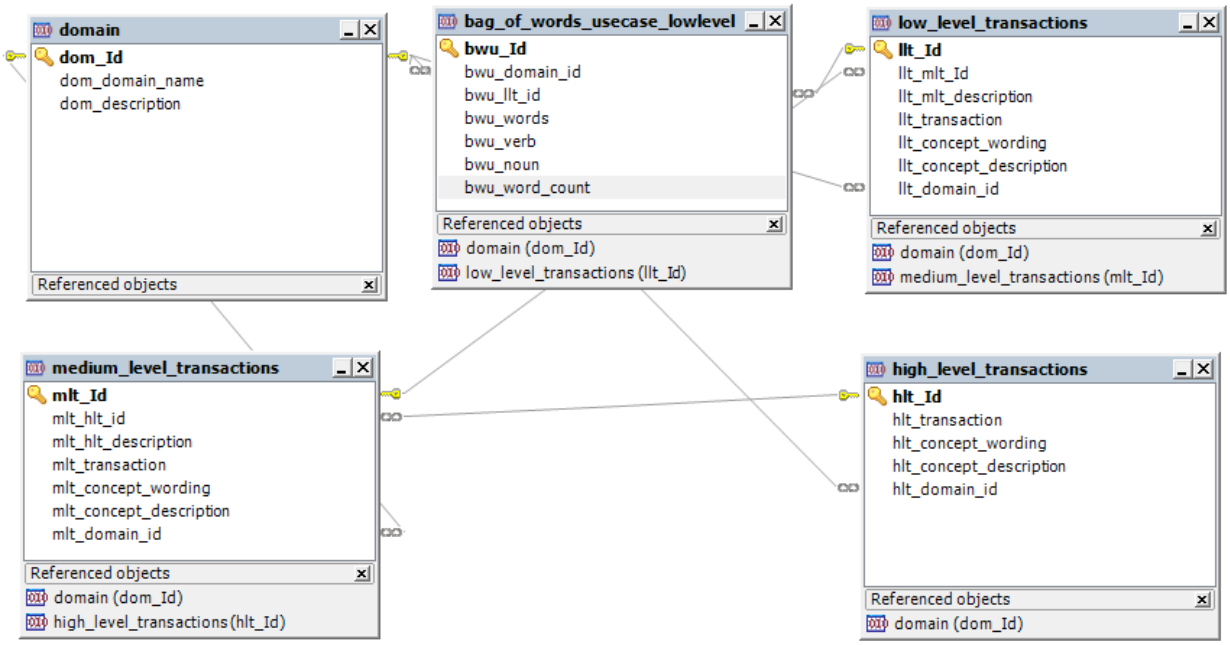
medium_level_transactions

Field *	Type *	Null *	Key *	Default	Extra *
mlt_Id	int(11)	NO	PRI	{null}	auto_increment
mlt_hlt_id	int(11)	YES	MUL	{null}	
mlt_hlt_description	varchar(255)	YES		{null}	
mlt_transaction	varchar(255)	YES		{null}	
mlt_concept_wording	varchar(255)	YES		{null}	
mlt_concept_description	varchar(255)	YES		{null}	
mlt_domain_id	int(11)	YES	MUL	{null}	

high_level_transactions

Field *	Type *	Null *	Key *	Default	Extra *
hlt_Id	int(11)	NO	PRI	{null}	auto_increment
hlt_transaction	varchar(255)	YES		{null}	
hlt_concept_wording	varchar(255)	YES		{null}	
hlt_concept_description	varchar(255)	YES		{null}	
hlt_domain_id	int(11)	YES	MUL	{null}	

All the related tables shown together:



5.5 Learning system

5.5.1 Ontology setup and use

After the class diagrams are obtained, it may happen that the class diagram contains little data and would need some additional input. At this stage the ontology is consulted, where there should be pre-populated data in the database for that domain. There are certain attributes that are marked as mandatory and these would, for example, include bank account number for a bank account class or a customer id for a customer class. Apart from these default attributes, the rest of the attributes and methods are added and they are each associated with two weights – a user defined weight and a system defined weight. The system defined weight is initially set in the database by the user and an incremental value must be decided before processing can start. Then, for each instance that the attribute is encountered, the system defined weight is increased by that the incremental value and

the maximum possible value is 1.0. Conversely every time a system defined attribute or method (which is not flagged as mandatory) is not encountered in the text, the system defined weight is decreased by the incremental value. The lowest value for any weight (user-defined or system-defined) is 0.0 and the maximum value is 1.0. The following formula is then applied to the various weights to compute a resulting value:

$$W = R \times W_{system} + (1 - R) \times W_{user}$$

The user needs to select a value of "R" which determines the ratio of user-defined and system-defined weight on the final computed value. If "R" is set to 1.0, only the system weight counts and if "R" is set to 0.0, only the user-defined is taken into account. If "R" is set to 0.5, then the user-defined weight and the system defined weight have equal contribution in the final computed value. Once the final value is computed, it can be compared to a threshold to validate if it would be accepted. A very high value for a threshold (like more than 0.85) would mean that the values from the ontology may not be considered and a very low threshold may mean that all the computed values would get accepted. The user needs to choose a threshold and the initial recommended value could be between 0.55 and 0.75. All these put together should produce a series of classes with attributes and methods.

When the class diagrams have been enhanced through the ontology, the output from the system is altered through the first mechanism of the learning system. The learning enhances the output from the text, with the use of an ontology. The weights are also adjusted meaning that data in the ontology is modified every time the system is run. For example, if an attribute is mentioned in the ontology and it is never encountered in the text, the system defined weight component would be reduced at each run until it falls below the threshold. The way the ontology is set up also allows for user validation of the data. If the ratio "R" is set to include exclusively user-defined data, then the user is always in control of the output. It would be better to have the output from the ontology as a mixture between user-defined data and system-defined data.

5.5.2 Neural network

An artificial neural network (ANN) is a network of software nodes, inspired by the biological functioning of human and animal brains, which are designed to allow a machine to learn a particular task over time. The idea is to allow a software program tackle a problem the way a

human brain would. ANN's have been applied to a variety of tasks including computer vision, speech recognition, machine translation, social network filtering, playing board and video games and medical diagnosis. The key features of a neural network were mentioned in the previous chapter and are not repeated here. There are various algorithms and ways to set up a neural network. The number of neurons, the activation function, the propagation function, the topology, the learning paradigms (supervised, unsupervised, self-learning), the organisation of the network (deep learning or not) and the network design are amongst the factors which contribute to the type of network. There are at least 25 popular and commonly used layouts for artificial neural networks ranging from a simple perceptron to a deep convolutional inverse graphics network and there are adaptations of these topologies and learning algorithm which are being tested and adopted by software engineers.

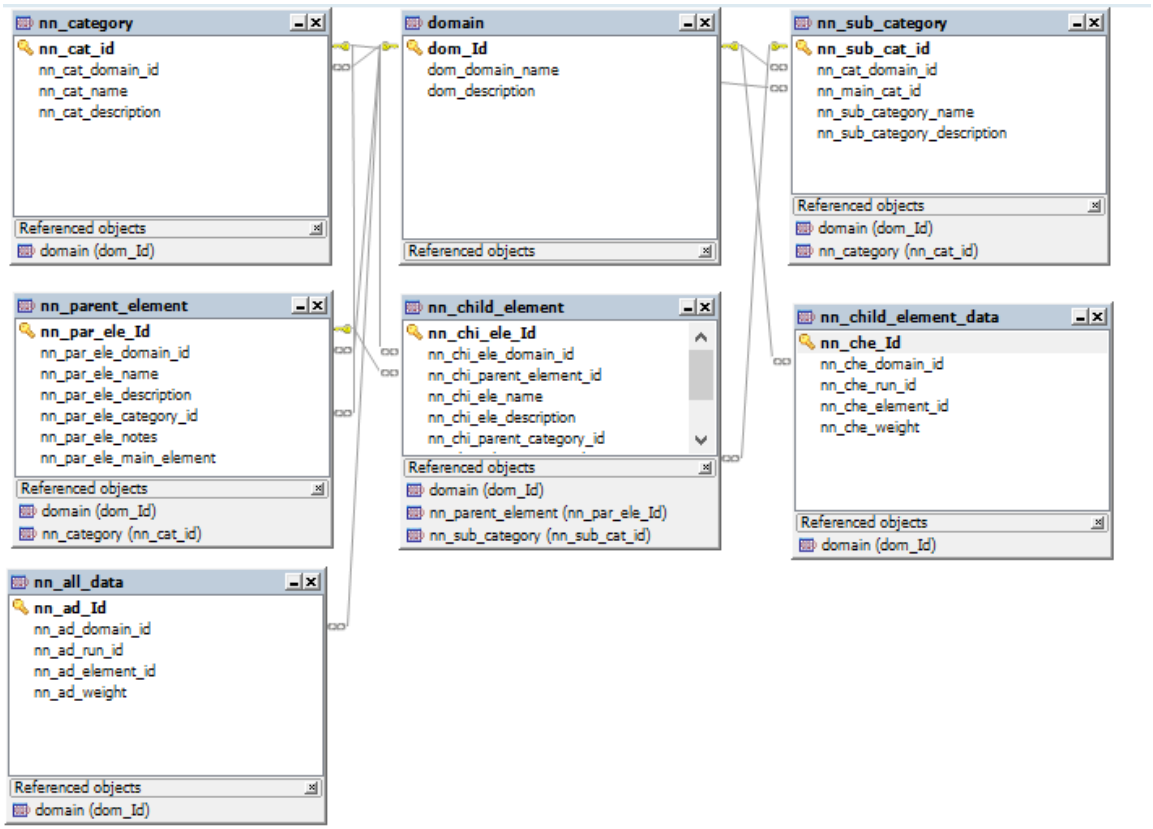
This research for the framework is not focused on neural networks as it is more focused on deriving a design from natural language requirements. However, it was also deemed necessary to have a learning system in place so that the framework would be different from existing approaches and could grow over time. It was decided to have a neural network help predict the design, in addition to the work being carried out by the ontology, and also use the neural network to abstract the design. Therefore, it was decided to have a simple neural network which is trained on time series prediction to predict the possible design and assist in the abstraction. Since the main focus of the research is not neural networks and it is rather elaboration of a design and architecture from natural language text, the neural network chosen would be a simple one. Time series prediction was used and it is based on historical data and could, in the future, be either paired up with the ontology data or use the ontology data for training purposes.

5.5.2.1 Neural network – implementation process

In order to predict the various aspects of the design, the key elements of the design artefacts were isolated in a similar manner to the ontology data. There would be two elements of the design in each case, one as a high-level element of the design and one as a low-level element of the design. Firstly, the data for the classes (high-level element) had to be entered in the backend. The same scale of values, between zero and one was selected for the historical values. In order for the time series prediction to work, there would have to be a list of historical values and the ANN would then use the historical values to predict a current value. The framework could once again use a

threshold value to decide if that element of the design is selected as part of the output. The data was split into two sets of values, a series of values for the high-level design elements and a series of values for the low-level design elements.

The data for the setup of the neural network was stored in the following tables in the database.



The Encog library was used to predict the time series and hence elements of the design. Encog is a lightweight and efficient neural network library available in Java which is also open-source. The version that was used was 3.3.0 and the libraries used were :

encog-core-3.3.0.jar

encog-core-3.3.0-javadoc.jar

encog-core-3.3.0-sources.jar

The neural network is built to consult the historical data in the database, where each high level element of the design (class or actor) should have a series of values, ranging from 0 to 1 (similar to the ontology). These historical values make up the data to be used for the time series prediction.

When the predicted value for a design element is computed, it is compared to a threshold (set to 0.7 initially) and the design element is accepted if the computed value exceeds that threshold. If the computed value from the neural network exceeds the threshold value, then the design element is accepted as part of the output from the neural network. If a high-level or parent design element is accepted as part of the design, then the algorithm goes on to compute if all the children element of that design element are to be accepted as well. For example, if the computed value, from the neural network, for a class exceeds the threshold and is accepted as part of the output, then all the attributes and methods of that class are processed, to check if they would make it as part of the design. For each child element, for example an attribute, the algorithm would retrieve all the historical data for that element and use the time series prediction facility from Encog to predict a value for that element. The predicted value is again compared to the threshold and it is included in the design if it exceeds that value. In that way, the neural network loops through all the design elements and eventually proposes a whole design, using the data in the backend as the starting point.

5.5.3 Neural network and design abstraction

The neural network is also used for the abstraction of the design. Once again, the time series prediction facility of the neural network is used. For each design element, that is present in the low-level design table, there is a series of associated values which make up the historical values for that design element. Similarly, the medium level design and the high-level design elements also have a series of values which are associated with each design element. The neural network goes through all of the existing values in order to predict if the current design element is to be included in the design. After working its way through all the design elements and deciding which ones to include, the neural network also predicts an abstracted design.

5.5.4 User feedback harnessing

The first way in which the user can provide feedback is through the ontology. The ontology is set up with data to begin so that the output from the system can be enhanced with the ontology data. The user would be allowed to intervene, after the program has been running, by importing an XML file into the system. The values in the XML file can reset all the values in the ontology, determine the ratio “R” which decides which portion of the ontology data is system defined or user defined,

and the incremental / decremental value by which the system weight needs to be increased or decreased if it is found in the text or not.

After the feedback is imported into the system, the future times the software is run the data would be used to affect the output from the ontology. The user can also set the ratio “R” so that the output from the ontology is exclusively that of the user-defined in the database.

5.6 Conclusion

The proposed framework was detailed and the main components outlined in the previous chapter. In this chapter, a high-level description of the implementation details of each component was given. The file is uploaded by the user and it is first processed by the NLP component. The NLP component uses the CORENLP libraries, the Stanford parser, the OPENNLP libraries, some custom logic and the database to clean, filter and process the text so that meaningful concepts can be extracted from the text. After the NLP component, the design extraction component uses the output from the NLP component to create the use cases and the class diagram. The learning system is then brought into contribution and it is made up on two units. The ontology uses the data stored in the backend and the concepts encountered in the text to propose additional features. The neural network is solicited to predict a design (use cases and class diagram) and also assist in abstracting the design into an architecture. The whole output from the framework is an XML file with the design and the architecture. The user can be solicited to provide some feedback by reviewing the data in the ontology and pre-populating the data for the neural network.

The technical implementation of the components of the framework was described in this chapter and the next step is to evaluate the results of the framework on a case study. The next chapter also provides a walkthrough of the detailed transformation of the input as it passes through each stage of processing, how the input data is converted into a design and how the neural network and the ontology are solicited to enhance the design and abstract it into an architecture. The output from the framework is also compared to the output that would be produced by a system designer and the two are compared and assessed. The accuracy, the automation and the impact of the framework are discussed.

CHAPTER 6: CASE STUDY AND FRAMEWORK EVALUATION

6.1 Introduction

After an analysis of the relative cost of defect leakages, requirement defects and existing approaches to reduce the introduction of defects into the design from the requirements, a framework was proposed. The previous chapter has provided a summary of the technologies and learning mechanisms which were used to design and implement the framework. The paramount question which remains is "How well can the framework deal with real world requirements?". In order to evaluate the framework, this chapter considers a case study. The case study is commonly used in object-oriented and service-oriented systems with a number of possible design approaches, and they are used to evaluate the framework. All the aspects of the framework are explained in details, with a walkthrough of how each step transforms the input and extracts a design from the requirements.

As part of the evaluation process, the output of the framework would be compared to a design, produced manually by a system designer. The goal is to compare the similarities and differences between a manual approach of how an architect would produce a design and the capacity of the framework to produce a design and an architecture which is distributed and suitable for service-oriented architecture (SOA) and micro-services architecture. The design and architecture produced by the framework should be suitable for a solution for the problem domain and problem specification.

This chapter starts with section 6.2, which provides a description of the first case study and a requirements analysis section. The requirements analysis corresponds to an analysis that would be carried out by a system designer (architect) when preparing to design the architecture of the case study manually. The key points highlighted in the requirements are retained and used when coming up with a design manually. Section 6.3 then presents all the settings needed for the framework to run as expected, including the information needed to set up the framework for a problem domain. Section 6.4 provides a walkthrough of how the NLP component is used to extract concepts and process the text for the case study. Section 6.5 explains how the design extraction component is

used to produce a design from the output of the NLP component. Section 6.6 details how the ontology is used to enhance the design produced so far. Section 6.7 explains the role of the neural networks and how it is used to abstract the design and also provides an overview of the abstracted design. Section 6.8 covers the output obtained by the framework and provides a brief explanation of how the results are to be interpreted. Section 6.9 explains the design obtained manually and the considerations taken into account when elaborating the design manually. Section 6.10 covers the evaluation of the framework by comparing the design from the manual approach to that of the framework. Finally, section 6.11 concludes the chapter.

6.2 Case study

The framework has been designed and implemented, backed by the research on various topics. However, in order to test how the framework is able to perform with real world requirements, it needs to be evaluated. In order to assess the performance of the framework, a case study was used to evaluate the output of the framework. It was decided that the case study would be within the financial domain as these applications involve multiple components, can be critical and warrant caution for a proper design and have been widely studied by academia. The particular case study chosen is the description of the functioning of the Automated Teller Machine (ATM) which describes how the customers and other stakeholders interact with the software system. The ATM is a widely studied case study and can be quite complex and can be designed using various types of architecture.

The choice of the case study is driven by the fact that the ATM provides a service that is communicating with a number of banking services such as authentication, transactions, reporting etc. within one bank as well as communication and obtaining services from other banks. Thus, ATM processes require communication between a number of components/services to complete a user request, including transaction, client (user interface) and back-end service as well as authentications etc. ATM, as a case study, has been commonly used in early object-oriented systems [167]. In addition, it exhibits the service concepts reflect in client/server architecture with the banking sector including different modules for front end (user interface) and back engine services. ATM services are relatively easy to model, and can be used as a proof of concept for evaluating the proposed framework. In addition, it can be modelled using different designs that broadly follow the Service Oriented Architecture (SOA) principles. Thus, it can be used for testing

different SOA architectural styles. The following text is the original unaltered text as it is uploaded into the system.

An ATM provides money to authorised users who have sufficient funds on deposit. It requires the user to provide a personal identification number (PIN) as an authorisation. Money is provided after a confirmation from the bank computer system. Overall the main function of the ATM is to provide a number of services to the customer:

- *A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.*
- *A customer must be able to make a cash withdrawal from any suitable account linked to his/her card. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or cheques in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator.*
- *A customer must be able to make a balance inquiry of any account linked to the card.*
- *A customer must be able to print the balance, mini-statements, receipts etc.*
- *Transfer money, change PINs etc.*

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a PIN. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned.

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc. The details of the customer such as customer name, customer age, customer address and salary are required.

6.2.1 Requirement analysis

In order to evaluate the results from the framework, the design produced for the case study would be compared with a design generated by a manual approach, just like an architect would come up with a design after reading and interpreting the requirements. The architect would derive key features from the requirements and consider them for the design. This section describes the key features important for the ATM system, which are to be considered when creating the design of an ATM system manually. The main features retained from the interpretation of the requirements are listed below:

An ATM provides money to authorised users who have sufficient funds on deposit. It requires the user to provide a personal identification number (PIN) as an authorisation. Money is provided after a confirmation from the bank's computer system. Overall, the main function of the ATM is to provide a number of services to the customer:

- A customer must be able to make a cash withdrawal from any suitable account linked to his/her card. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or cheques in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator.
- A customer must be able to make a balance inquiry of any account linked to the card.
- A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.
- A customer must be able to print the balance, mini-statements, receipts etc.
- Transfer money, change PINs etc.

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a PIN. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned.

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify

and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.

6.3 Experimental setting

Prior to using the framework, there are a few configuration and settings which need to be in place so that the user may use the framework and yield the best result out of it. Firstly, the database must be set up accordingly. A SQL script is used to create all the necessary database tables and all the primary entries in those tables are used by the framework. Some details were provided on the individual tables in the previous chapter and there are more details of some of the other tables provided throughout this chapter.

The database is used for two main purposes. Firstly, some of the tables are set up and used to allow the framework to run in general (NLP component and design extraction component) while the second purpose is to assist the learning system and the prediction of future values. There are 21 tables in the database and they all need to be created prior to the software running in order to avoid runtime errors. The next paragraph details how the database is used for the general running of the framework and the NLP parsers.

The most important table is the domain table. The data for the framework is to be arranged by domain where a domain is the field in which the requirements are intended for. The domain is used to group, classify and organize the data for the learning system and other tables, which have a support function, depend on the domain table. It would be very hard, if not impossible, to have a learning system with data encompassing all the industries in existence. Therefore, there is a domain table which is used to define a series of domain to segregate the knowledge of the ontology and the learning system by domains. A domain can be very subjective and in order for the learning system of the framework to be efficient and purposeful, it is recommended that a domain be quite specific. For example, a domain defined as “Banking” may be very vague and would encompass the whole banking world. A more specific area of the “banking” domain would be more desirable, such as “ATM – configuration for an ATM system”. Therefore, within the Banking field, several domains could be defined. The following picture shows the entries in the domain table as it was set up to run the program.

dom_Id *	dom_domain_name	dom_description
1	Software for mobile	Covers the set for mobile software and hardware
2	Bank - ATM	Covers all the bank related transactions for an ATM machine

Figure 6.1: Domain table sample data

For each defined domain, the following tables are then used to create an ontology: classes, class_attributes and class_methods. For each domain a series of classes have to be defined and for each class, attributes and methods also have to be enumerated so that the ontology can enhance the design. In case the classes, attributes and methods are not defined for a particular domain, the ontology would not really be able to enhance the design produced by the framework. The setup script provided, inserts values in the above-mentioned tables for the "ATM" domain. It is important to know that the framework would also insert and maintain values in the three tables used for the ontology after the first run. However, it is recommended that the user also pre-populates the tables in order to ensure that key elements of the design are included in the final output derived from the framework.

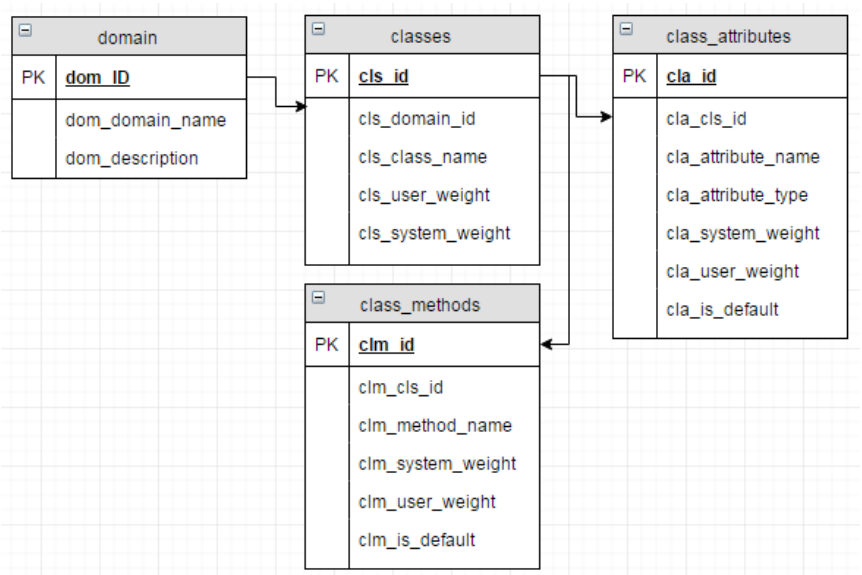


Figure 6.2: Tables for the ontology

The next set of tables to be set up with data are some of the tables used by the NLP component. They are the "synonyms" and "special_character" tables. The "special_character" table is used to replace special characters by a substitution text, allowing the user to replace unusable and undesired characters by a more meaningful text or simply blank text. Other special characters like bullet points or unknown characters which may be created when the framework is attempting to

read a document in a particular format and converting that document to a text file, can be eliminated using that table. The user can maintain this table and it would be domain independent. The next table is the "synonyms" which is used to replace a word or an acronym by a text. This table may also be used for the reduction of concepts and that is explained later in the chapter. These tables are used by the NLP component and section 6.4 provides a step-by-step walkthrough of the data from these tables are used by the NLP component to transform the input.

6.3.1 Other settings

The other settings detail the environment setting used to test the software. As mentioned in the previous chapter, there are a series of external libraries which are used and they are summarised in the table below. Java 1.8 is needed on the computer running the framework and the following external libraries are also required to run the application. All of the libraries mentioned in the previous chapter are listed below, with the addition of the MYSQL Connector, which is needed by the framework to connect to the MYSQL database.

JAR Files required	Version	Library Name
Stanford Core NLP	3.6	stanford-corenlp-3.6.0.jar
		stanford-corenlp-3.6.0-javadoc.jar
		stanford-corenlp-3.6.0-models.jar
		stanford-corenlp-3.6.0-sources.jar
Stanford parser	3.4	stanford-parser.jar
		stanford-parser-3.4-javadoc.jar
		stanford-parser-3.4-models.jar
		stanford-parser-3.4-sources.jar
Open NLP	1.6	opennlp-tools-1.6.0.jar
		opennlp-tools-1.6.0-sources.jar
MY SQL Connector	5.1.6	mysql-connector-java-5.1.6.jar
Encog	3.30	encog-core-3.3.0.jar
		encog-core-3.3.0-javadoc.jar
		encog-core-3.3.0-sources.jar

Table 6.1: External Java libraries needed for the framework

The next section provides a step-by-step walkthrough of how the different components transform the input data and eventually produces a design from the text.

6.4 NLP component

As mentioned in the previous chapter, the first component to interact with the data is the natural language processing component. The component is made up of six main steps and the following sections provide a step-by-step walkthrough of a case study and how the input is transformed throughout the NLP component.

6.4.1 Eliminating special characters

When the text file is uploaded into the framework and passed onto the parser, there are a series of special characters which may be introduced, especially when a word document (.doc,.docx extension) file is uploaded. The special characters mainly occur when characters like bullet points are interpreted by the parser. In order to remove such special characters, the framework replaces the special characters by another more meaningful value or simply a blank. All of these features are driven by using lookup database table "special_character". When the text was loaded from the text file, the text available for processing in the data structure was the following one:

ï»¿An ATM provides money to authorised users who have sufficient funds on deposit. It requires the user to provide a personal identification number (PIN) as an authorisation. Money is provided after a confirmation from the bank computer system. Overall the main function of the ATM is to provide a number of services to the customer: â€¢ A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.â€¢ A customer must be able to make a cash withdrawal from any suitable account linked to his/her card. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or cheques in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator.â€¢ A customer must be able to make a balance inquiry of any account linked to the card. â€¢ A customer must be able to print the balance, mini-statements, receipts etc.â€¢ Transfer money, change PINs etc.The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a PIN. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the

customer indicates that he/she desires no further transactions, at which point it will be returned.The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.The details of the customer such as customer name, customer age, customer address and salary.

All the special characters injected in the text are highlighted in yellow in the above text. Therefore, in order to remove these special characters and return the text to a more acceptable and readable version of the requirements, the values of the “special_characters” are used to replace all the special characters. The values in the special_characters table created so far are shown below:

spc_Id *	spc_character	spc_replacement_text
2	•	" "
3	â€¢	" "
4	¿	
5	¡	

Figure 6.3: Special characters table data

Any other special characters encountered in the future can be specified in this table. The replacement of special characters is a default operation and will always happen regardless of the domain (not a domain specific operation). The aim is to prevent special characters from hindering the performance of the parsers and the learning system at a later stage.

6.4.2 Coreference resolution

Coreference resolution is the process of designating the words which are being referred to, by one or many other words in the same text. A simple sentence may contain a subject, and that subject is then referenced by another word which may or may not be in the same sentence. In order to avoid referencing the same entity by two or more words, coreference resolution is used as a means to reduce the number of concepts in the text. Thus, the design is not burdened or made bulky by multiple terms which point to the same entity.

As mentioned in the previous chapter, the Core NLP library from the Stanford Natural Language Processing Group was used for Coreference resolution. The version 3.6.0 was downloaded and used for the needs of this research. The coreference capability of the tool was applied on the text and additional coding was done in order to ensure that only 25 of the most common pronouns were being replaced. The algorithm may get bulky and include quite a bit of self-reference and again the codes were customized in order to ensure that self-references were ignored. After this step, the text was transformed to the following:

" An ATM provides money to authorised users who have sufficient funds on deposit . An ATM requires the user to provide a personal identification number -LRB- PIN -RRB- as an authorisation . Money is provided after a confirmation from the bank computer system . Overall the main function of the ATM is to provide a number of services to the customer : " " A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine . " " A customer must be able to make a cash withdrawal from any suitable account linked to his/her card . A customer must be able to make a deposit to any account linked to the card , consisting of cash and/or cheques in an envelope . The customer will enter the amount of the deposit into the ATM , subject to manual verification when the envelope is removed from the machine by an operator . " " A customer must be able to make a balance inquiry of any account linked to the card . " " A customer must be able to print the balance , mini-statements , receipts etc. " " Transfer money , change PINs etc.The ATM will service one customer at a time . A customer will be required to insert an ATM card and enter a PIN . The customer will then be able to perform one or more transactions . The card will be retained in the machine until the customer indicates that he/she desires no further transactions , at which point An ATM will be returned.The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers . After turning the switch to the `` on " position , the operator will be required to verify and enter the total cash on hand . The machine can only be turned off when the machine is not servicing a customer . When the switch is moved to the `` off " position , the machine will shut down , so that the operator may remove deposit envelopes and reload the machine with cash , blank receipts , etc.The details of the customer such as customer name , customer age , customer address and salary .

The three corrections made by the coreference resolution algorithm are highlighted in the text above. The text highlighted in red corresponds to an inaccurate substitution. Given that the coreference algorithm is from Stanford university, it was decided to leave the result as it is. An updated version of the library can be downloaded in the future and the same inefficiencies can be checked for, in the future.

6.4.3 Isolating sentences

In order to ensure that the input text is accurately parsed, it was decided that the input text would be parsed by the OPENNLP parser so that the sentences can be isolated. This is to avoid lengthy sentences being amalgamated into one unit, which may hinder the rest of the parsing process and affect the concepts obtained from the text. For this particular case study, there is no significant change as the sentences were correctly segregated. After the sentences are isolated by the parser, the text looks as follows:

" An ATM provides money to authorised Customer who have sufficient funds on deposit .
An ATM requires the user to provide a personal identification number -LRB- PIN -RRB- as an authorisation .
Money is provided after a confirmation from the bank computer system .
Overall the main function of the ATM is to provide a number of services to the customer : " " A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine .
" "

A customer must be able to make a cash withdrawal from any suitable account linked to his/her card .
A customer must be able to make a deposit to any account linked to the card , consisting of cash and/or cheques in an envelope .
The customer will enter the amount of the deposit into the ATM , subject to manual verification when the envelope is removed from the machine by an operator .
" "

A customer must be able to make a balance inquiry of any account linked to the card .
" "

A customer must be able to print the balance , mini-statements , receipts etc .

" " Transfer money , change PINs etc .

The ATM will service one customer at a time .

A customer will be required to insert an ATM card and enter a PIN .

The customer will then be able to perform one or more transactions .

The card will be retained in the machine until the customer indicates that he/she desires no further transactions , at which point An ATM will be returned .

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of Customer .

After turning the switch to the `` on " position , the operator will be required to verify and enter the total cash on hand .

The machine can only be turned off when the machine is not servicing a customer .

When the switch is moved to the `` off " position , the machine will shut down , so that the operator may remove deposit envelopes and reload the machine with cash , blank receipts , etc .

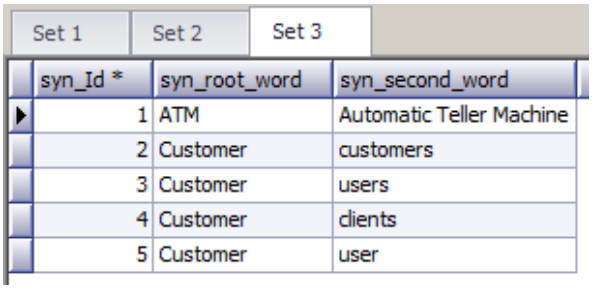
The details of the customer such as customer name , customer age , customer address and salary.

6.4.4 Reduction of concepts

The next step is then to replace the synonyms and the database table “synonyms” is used to that effect. The entries to be made in this table may or may not correspond to the traditional English synonyms, as it is commonly understood. The aim here is to have words which correspond to the same concept and should be grouped as one concept. For now, this table is also designed not to be domain specific, but it is planned to be domain-specific in the future. This means that the substitution of the words would apply for all domains.

The entries can be used to replace / handle plurals also. There was another algorithm coded to handle plurals, but it has been parked for now. The reason is that in some cases, the words which are created as a result of the algorithm may not correspond to a real English word. There may also be technical terms which are added to the text such as the words “apps” or “ATMs” and it becomes tedious to reduce such words to the singular form.

The entries in the synonyms table are as follows:



Set 1 | Set 2 | Set 3

syn_Id *	syn_root_word	syn_second_word
1	ATM	Automatic Teller Machine
2	Customer	customers
3	Customer	users
4	Customer	clients
5	Customer	user

Figure 6.4: Synonyms table data

All the terms present in the text and which are present in the “syn_second_word” are then replaced by the word in the column “syn_root_word”. After this step the text looks as follows:

" An ATM provides money to authorised Customer who have sufficient funds on deposit .
An ATM requires the Customer to provide a personal identification number -LRB- PIN -RRB- as an authorisation .
Money is provided after a confirmation from the bank computer system .
Overall the main function of the ATM is to provide a number of services to the customer : " " A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine .
" "
A customer must be able to make a cash withdrawal from any suitable account linked to his/her card .
A customer must be able to make a deposit to any account linked to the card , consisting of cash and/or cheques in an envelope .
The customer will enter the amount of the deposit into the ATM , subject to manual verification when the envelope is removed from the machine by an operator .
" "
A customer must be able to make a balance inquiry of any account linked to the card .
" "
A customer must be able to print the balance , mini-statements , receipts etc .
" " Transfer money , change PINs etc .
The ATM will service one customer at a time .
A customer will be required to insert an ATM card and enter a PIN .
The customer will then be able to perform one or more transactions .

The card will be retained in the machine until the customer indicates that he/she desires no further transactions , at which point An ATM will be returned .
The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of Customer .
After turning the switch to the `` on '' position , the operator will be required to verify and enter the total cash on hand .
The machine can only be turned off when the machine is not servicing a customer .
When the switch is moved to the `` off '' position , the machine will shut down , so that the operator may remove deposit envelopes and reload the machine with cash , blank receipts , etc .
The details of the customer such as customer name , customer age , customer address and salary.

The aim is to avoid having words like “customers”, “customer”, “user” and “clients” as four different concepts, which would eventually translate to classes. Thus, redundant elements of the design can be grouped, in order to avoid inefficiency.

6.4.5 NLP processing rules

Once the text has been cleaned and pre-processed a few times by the above-mentioned steps, it is now time for the parser to come into play. The text is now processed by the parser and the parsed data is available in data structures of the Stanford NLP parser. Initially, the algorithm simply stores all the words encountered in the text into the “word_count” table. The count (number of times a word is encountered in the text) is kept and this data is latter used to compute the frequency of each word. This information will be used to determine terms which are believed to be concepts.

When all the words in the text have been logged in the database, the rules described in section 5.2.1.5 (NLP Processing rules) are applied in order to identify concepts and store them in the database and some data-structures for further processing. The rules applied are listed and the data obtained from the text for each step is also mentioned.

- Identify the last word of each sentence. Save these words in the list Stop words. The stop words identified are: *authorization, system, card, envelope, etc, time, PIN, transactions, returned, Customer, hand, customer, salary.*
- Keep track of the total number of words in the text. Also keep a running count of the number of occurrences of each word.

- Calculate the frequency of each word. The number of times each word occurs and the frequency of each word is stored in the word_count table. The following shows a sample of the word count and frequency.

wrd_Id *	wrd_Word	wrd_Count	wrd_Frequency	wrd_TotalCount	wrd_run_Id
3951	ATM	8	0.0219	366	46
3952	provides	1	0.0027	366	46
3953	money	3	0.0082	366	46
3954	to	21	0.0574	366	46
3955	authorised	1	0.0027	366	46
3956	Customer	18	0.0492	366	46

- Use an NLP (Stanford) parser to parse the whole document. This is done and the parser returns the data in a parse tree, a data structure custom to Stanford parser, and the results are saved for later use.

- Store all the nouns in a list, the Nouns list. The noun list was made up of the following terms: *ATM, money, Customer, funds, deposit, user, identification, number, PIN, authorization, Money, confirmation, bank, computer, system, function, services, customer, transaction, progress, key, request, machine, cash, withdrawal, account, his/her, card, envelope, amount, verification, operator, balance, inquiry, mini-statements, receipts, PINs, etc, time, transactions, desires, point, switch, servicing, position, hand, envelopes, details, name, age, address, salary.*

- All the nouns are saved in the concepts list.

- All the subjects of the verbs (regardless of whether they are nouns) are also added to the concepts list.

The whole text uploaded for a particular file, is broken into words and stored in the “word_count” table. After the above-mentioned processing rules are applied and all the concepts are applied, then the words identified as concepts are stored in the concept_word table. All the words identified as concepts are then retained for later processing and mainly for the design derivation. The next sections describe how the design artefacts are derived from the text and how the learning system is brought into contribution to improve the design.

6.4.6 Concepts identification and elaboration

After all the concepts have been identified from the NLP processing rules, they are further filtered according to the rules below:

- All the concepts (coming from the noun list) which have a frequency of less than 2 % are ignored for future processing.
- All the subjects of the verbs are considered for future processing, regardless of frequency.
- Words which are too generic, or correspond to a design element, a proper name or a geographical location are ignored for future processing. The database stores a list of words which may be ignored as they are irrelevant or too generic. For example, words like “machine”, “function”, “London”, “John”, etc… can be added to that table so that they are ignored.

The final lists of concepts retained for further processing are: *ATM, money, funds, user, identification, number, Money, confirmation, bank, computer, function, services, transaction, progress, key, request, machine, cash, withdrawal, account, his/her, amount, verification, operator, balance, inquiry, mini-statements, receipts, PINs, desires, point, switch, servicing, position, envelopes, details, name, age, address.*

6.5 Design extraction component

After the data has been cleaned, filtered and altered by the NLP component, the design extraction component uses the output of the NLP component to create the design and the architecture. This section provides a walkthrough of how each of the design elements and the architecture are produced. The first elements of the design to be created are the use cases.

6.5.1 Use case derivation

All the words which have been identified as concepts and which are the subject of a verb are considered as actors. The aim here is to identify as many relevant use cases as possible. Verbs not in the present tense (such as past participle etc…) are ignored and all the other use cases for an actor are recorded in a data structure. The use cases are created for one sentence at a time and are grouped for each actor. The resulting group of use cases are then scanned and filtered to avoid duplicate or similar use cases.

Firstly, the data structure is filtered to allow only a single occurrence of each use case and avoid duplicate use cases. Secondly all the verbs described in the use case are scanned in order to ensure that they are pointing to a direct object. Any use case with a verb not pointing to a direct object will be ignored. Lastly the result of the parser is again scanned so that all the verbs which have a similar behaviour and are pointing to the same subject and object are also considered only once. For example, for a sentence like:

A customer must be able to make a balance inquiry of any account linked to the card.

The algorithm described so far, is most likely going to identify two use cases, where one describes the verb as "able" and the second identifying the word "make" as the verb. In this case, this step will clean the output so that only one use case is used for the final output. The same series of step are repeated for all the remaining sentences. At the end of it, the use cases are produced for the whole document.

For the requirement set shown earlier the following use cases were identified.

Actor	Action / Use case
ATM	provides money to authorised Customer who have sufficient funds on deposit .
	requires the Customer to provide a personal identification number (PIN) as an authorisation .
	service one customer at a time .
	have a key-operated switch that will allow an operator to start and stop the servicing of Customer .

Table 6.2: Use cases for *ATM* actor in the case study

Actor	Action / Use case
Customer	have sufficient funds on deposit .
	abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine . " "
	make a cash withdrawal from any suitable account linked to his/her card .
	make a deposit to any account linked to the card , consisting of cash and/or cheques in an envelope .
	enter the amount of the deposit into the ATM , subject to manual verification when the envelope is removed from the machine by an operator . " "
	make a balance inquiry of any account linked to the card . " "
	print the balance , mini-statements , receipts etc . " "

	insert an ATM card and enter a PIN .
	perform one or more transactions .
	indicates that he/she desires no further transactions , at which point An ATM will be returned .

Table 6.3: Use cases for *Customer* actor in the case study

Actor	Action / Use case
switch	allow an operator to start and stop the servicing of Customer .

Table 6.4: Use cases for *switch* actor in the case study

Actor	Action / Use case
operator	verify and enter the total cash on hand .
	remove deposit envelopes and reload the machine with cash , blank receipts , etc .

Table 6.5: Use cases for *operator* actor in the case study

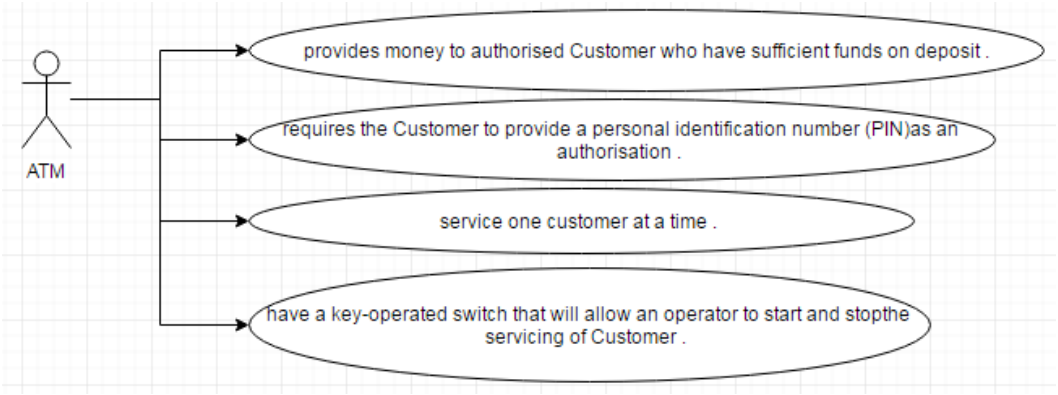


Figure 6.5: Use cases for actor "ATM"

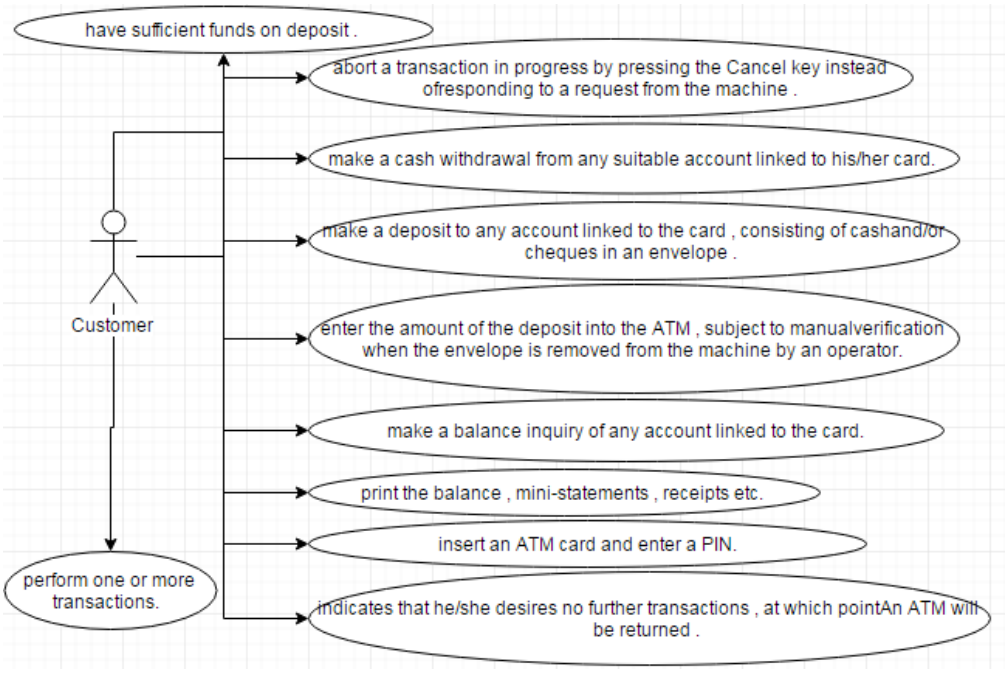


Figure 6.6: Use cases for actor "Customer"

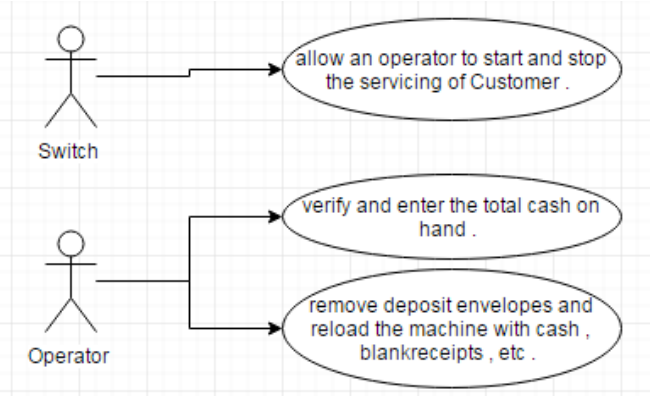


Figure 6.7: Use cases for actors "Switch" and "Operator"

All the use cases are created in data structures and have been drawn manually in a UML editor for the sake of this report.

6.5.2 Class diagram extraction

The next step is to create the class diagrams from the text and the knowledge extracted from the text. The use cases and the algorithm used to produce the use cases is used as a starting point. The actors of all the use cases are stored as classes. The verbs of the use cases and the direct objects of these verbs are used to derive the methods of these verbs. The next step is to derive the attributes and the following rule is used to derive the attributes:

If there are two consecutive nouns in the text and the first one is a class then the second noun is considered as an attribute of that class.

After the steps described above the following classes, methods and attributes were found.

Class	Methods	Attributes
ATM	provides money	
	requires Customer	
	service customer	
	have switch	

Table 6.6: Methods and attributes of the class *ATM*

Class	Methods	Attributes
Customer	have funds	Name
	make deposit	Age
	enter amount	address
	make inquiry	
	abort transaction	
	print balance	
	insert card	
	enter PIN	
	perform transactions	
	indicates transactions	

Table 6.7: Methods and attributes of the class *Customer*

Class	Methods	Attributes
switch	allow operator	

Table 6.8: Methods and attributes of the class *switch*

Class	Methods	Attributes
operator	verify cash	
	remove envelopes	
	reload machine	

Table 6.9: Methods and attributes of the class *operator*

The class diagram generated by the system would look as follows, when drawn in an OOP design tool.

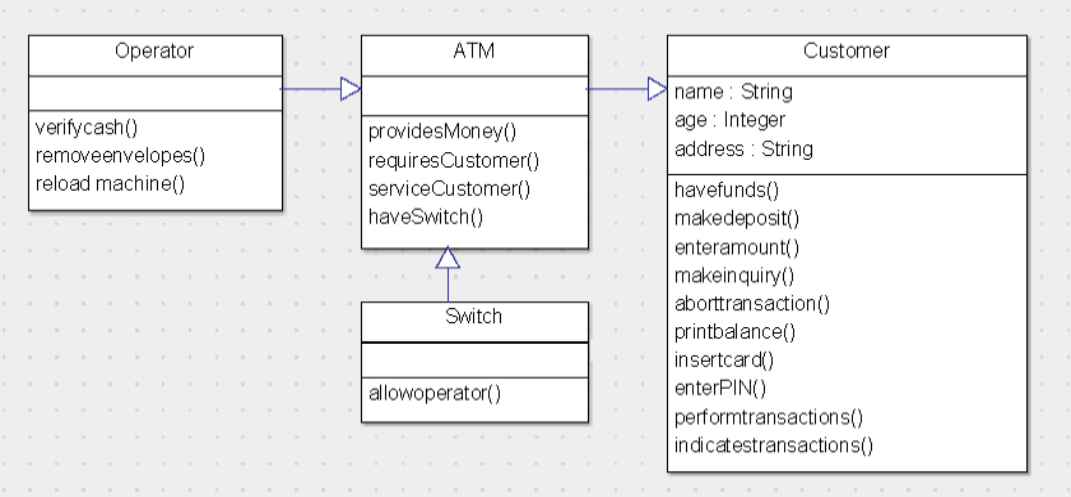


Figure 6.8: Class diagram from design extraction component

The classes, attributes and methods produced above can be further enhanced by using the ontology. The next section describes how the ontology and the neural networks are brought into contribution to enhance the output.

6.6 Application of the learning system

The learning system comprises of an ontology and a neural network. The ontology is used to add supplementary elements from the database to the class diagram and the use cases. The ontology is configured so that it is made up of a series of values which represent a system component and a user component. The ontology is chosen because it permits the user to influence on the output. The ontology can be configured to use the user entered data or the system data in any ratio between 0 to 100%. When the user data is used at 100%, there is no input from system data and vice versa. Table 6.12 to Table 6.15 demonstrate how the ontology enhances the output from the framework. The elements added in dark blue come from the ontology data and algorithm.

The neural network is used to abstract the design. The design obtained from the text is considered as a low-level design. For a series of words, a corresponding element of a low-level design is configured in the backend. This technique is inspired by the bag-of-words parsing technique. The design elements are expressed as a series of number and a Time series prediction model is used to determine the design element that each low-level design artefact abstracts to. The process is repeated three times so that the abstracted design is obtained. Table 6.17 shows the output from the neural network.

6.7 Enhancing output with ontology data

The classes, attributes and methods produced above can be further enhanced by using the ontology. The tables used for the ontology have been described in the previous sections and they are the "classes", "class_attributes" and "class_methods" tables. The data present in these tables are then used by the framework to enhance the class diagrams. For each of the table there are two columns which are used to compute the output data. They are the user weight and the system weight and the data listed in the tables below were pre-populated for a given domain. The data needs to be seeded so that the program can use the user weight and the system to compute a weight for each element of design and decide if that can be included as the output of the ontology. The tables were populated as follows:

cls_Id *	cls_domain_id	cls_class_name	cls_user_weight *	cls_system_weight *
1	2	ATM	0.5	0.99
2	2	Customer	0.5	0.99
3	2	Card	0.5	0.83
4	2	Account	0.5	0.83
5	2	Bank	0.5	0.83
7	2	switch	0.85	0.92
8	2	operator	0.85	0.92

Figure 6.9: Data in *Class* table (used by the ontology)

cla_Id	cla_cls_id	cla_attribute_name	cla_attribute_type	cla_system_weight	cla_user_weight	cla_is_default
1	1	'ATM_id'	'int'	0.85	0.5	1
2	1	'ATM_balance'	'double'	0.85	0.5	1
3	1	'ATM_location_id'	'int'	0.85	0.5	1
4	2	'Customer_id'	'int'	0.85	0.5	1
5	2	'Customer Name'	'string'	0.83	0.5	0
6	2	'Address'	'string'	0.92	0.5	0
7	2	'Phone Number'	'int'	0.83	0.5	0
8	2	'Date of Birth'	'int'	0.83	0.5	0
9	3	'Card_id'	'int'	0.85	0.5	1
10	3	'Account_id'	'int'	0.85	0.5	1
11	3	'Card Number'	'int'	0.85	0.5	0
12	3	'Expiry Date'	'date'	0.85	0.5	0
13	3	'Card Type'	'string'	0.85	0.5	0
14	3	'Card Limit'	'double'	0.85	0.5	0
15	4	'Account_id'	'int'	0.85	0.5	1
16	4	'Customer_id'	'int'	0.85	0.5	1

17	4	'Card_id'	'int'	0.85	0.5	1
18	4	'Account Number'	'int'	0.85	0.5	0
19	4	'Account type'	'string'	0.85	0.5	0
20	4	'Balance'	'double'	0.85	0.5	0
21	4	'Interest rate'	'double'	0.85	0.5	0
22	5	'Bank Name'	'string'	0.85	0.5	1
23	5	'Branch_id'	'int'	0.85	0.5	1
24	5	'Branch Address'	'int'	0.85	0.5	0
25	5	'Branch Phone Number'	'int'	0.85	0.5	0
26	5	'Branch Manager'	'int'	0.85	0.5	0
27	1	'card'	NULL	0.93	0.85	0
28	2	'name'	NULL	0.91	0.85	0
29	2	'age'	NULL	0.91	0.85	0

Table 6.10: Data in class_attribute table

clm_Id	clm_cls_id	clm_method_name	clm_system_weight	clm_user_weight	clm_is_default
1	1	Accept Card	0.83	0.5	0
4	1	Return Card	0.83	0.5	0
5	1	Request PIN	0.83	0.5	0
6	1	Validate PIN	0.83	0.5	0
7	1	Change PIN	0.83	0.5	0
8	1	Dispense Cash	0.83	0.5	0
9	1	Abort Transaction	0.83	0.5	0
10	1	Print Receipt	0.83	0.5	0
11	1	Transfer Money	0.83	0.5	0
12	2	Open Account	0.83	0.5	0
13	2	Make Deposit	0.99	0.5	0
14	2	Enquire Balance	0.83	0.5	0
15	2	Enter Pin	0.99	0.5	0
16	2	Change PIN	0.83	0.5	0
17	2	Take Cash	0.83	0.5	0
18	2	Accept Card	0.83	0.5	0
19	2	Withdraw Money	0.83	0.5	0
20	3	Check Balance	0.85	0.5	0
21	3	Expire	0.85	0.5	0
22	4	Calculate Interest	0.85	0.5	0
23	4	Update Balance	0.85	0.5	0
24	4	Provide Balance	0.85	0.5	0
25	5	Create Account	0.85	0.5	0
26	5	Issue Cards	0.85	0.5	0
27	5	Transfer Money	0.85	0.5	0

28	5	Accept Customers	0.85	0.5	0
29	5	Hold Accounts	0.85	0.5	0
30	1	provides money	0.9	0.85	0
31	1	requires Customer	0.9	0.85	0
32	1	service customer	0.9	0.85	0
33	1	have switch	0.9	0.85	0
34	2	have funds	0.9	0.85	0
35	2	abort transaction	0.9	0.85	0
36	2	make withdrawal	0.9	0.85	0
37	2	enter amount	0.9	0.85	0
38	2	make inquiry	0.9	0.85	0
39	2	print balance	0.9	0.85	0
40	2	insert card	0.9	0.85	0
41	2	perform transactions	0.9	0.85	0
42	2	indicates transactions	0.9	0.85	0
43	7	allow operator	0.9	0.85	0
44	8	verify cash	0.9	0.85	0
45	8	remove envelopes	0.9	0.85	0
46	8	reload machine	0.9	0.85	0

Table 6.11: Data in class_method table

There is a lot of data above and it is hard to understand how all this would translate into a design. So, this paragraph provides a walkthrough of how the data is used to enhance the design. The classes which occur both in the ontology and identified by the NLP parser are “ATM” and “Customer”. The data for these two classes are enhanced with the input from the ontology. The “cla_is_default” column is used to denote a default column (mandatory) column for the class, regardless of the weight computation. This is usually used to point to a column such as an identifier column. For example, every ATM class would need to have an ATM_id to uniquely identify an instance of an ATM. It is very likely that the fields such as the ATM_id, or any identifier field would not be explicitly mentioned in the text. Similarly, the field “clm_is_default” is used to denote mandatory or compulsory methods of a class in the table “class_method”.

The weightage is computed to decide if other (non-default) attributes and methods are to be added to the design. So, to remind ourselves, the weight is computed by the following formula:

$$W = R \times W_{system} + (1 - R) \times W_{user}$$

R stands for the ratio and it is set to a value of 0.75, which implies that 75% of the weight would constitute the weight of the system and 25% of the weight is picked from the user defined weight. The weight is computed and then compared with a threshold value. In this case, the threshold value is set to 0.75. Therefore, the computed weight would need to exceed that value to be included in the design.

clm_Id *	clm_cls_id	clm_method_name	clm_system_weight *	clm_user_weight *	clm_is_default
1	1	Accept Card	0.81	0.65	0
4	1	Return Card	0.81	0.65	0
5	1	Request PIN	0.81	0.65	0

Let's consider the three above entries which are linked to the class "ATM". Each entry has the system weight assigned to 0.81 and the user weight assigned to 0.65. The ratio R is set to that 0.75, meaning 75% of the calculated value comes from the system weight and user weight accounts for 25%. The computed ratio is (0.75 * 0.81) + ((1 – 0.75) * 0.65) = 0.6075 + 0.1625 = 0.77. The computed weight is therefore 0.77, and this exceeds the threshold value of 0.75. So, the values in the ontology are then picked by the framework and added to the design. After the values of the ontology are used to enhance the design, the class were as follows (values in dark blue are pulled from the ontology):

Class	Methods	Attributes
ATM	provides money	ATM_id
	requires Customer	ATM_balance
	service customer	ATM_location_id
	have switch	
	Accept Card	
	Return Card	
	Request PIN	
	Validate PIN	
	Change PIN	
	Dispense Cash	
	Abort Transaction	
	Print Receipt	
	Transfer Money	

Table 6.12: Methods and attributes of the class *ATM*, after enhancement by ontology data

Class	Methods	Attributes
Customer	have funds	name
	make deposit	age
	enter amount	address
	make inquiry	Customer_id
	abort transaction	Customer Name
	print balance	Address
	insert card	Phone Number
	enter PIN	Date of Birth
	perform transactions	
	indicates transactions	
	Open Account	
	Make Deposit	
	Enquire Balance	
	Enter Pin	
	Change PIN	
	Take Cash	
	Accept Card	
	Withdraw Money	

Table 6.13: Methods and attributes of the class *ATM*, after enhancement by ontology data

Class	Methods	Attributes
switch	allow operator	

Table 6.14: Methods and attributes of the class *switch*, after enhancement by ontology data

Class	Methods	Attributes
operator	verify cash	
	remove envelopes	
	reload machine	

Table 6.15: Methods and attributes of the class *switch*, after enhancement by ontology data

Class	Methods	Attributes
card	Check Balance	Card_id
	Expire	Account_id
		Card Number
		Expiry Date
		Card Type
		Card Limit

Table 6.16: Methods and attributes of the class *card*, which is added by the ontology

When the ontology data has been added to the output, the class diagram would look as follows. It is to be noted that the class “Card” is added completely from the ontology.

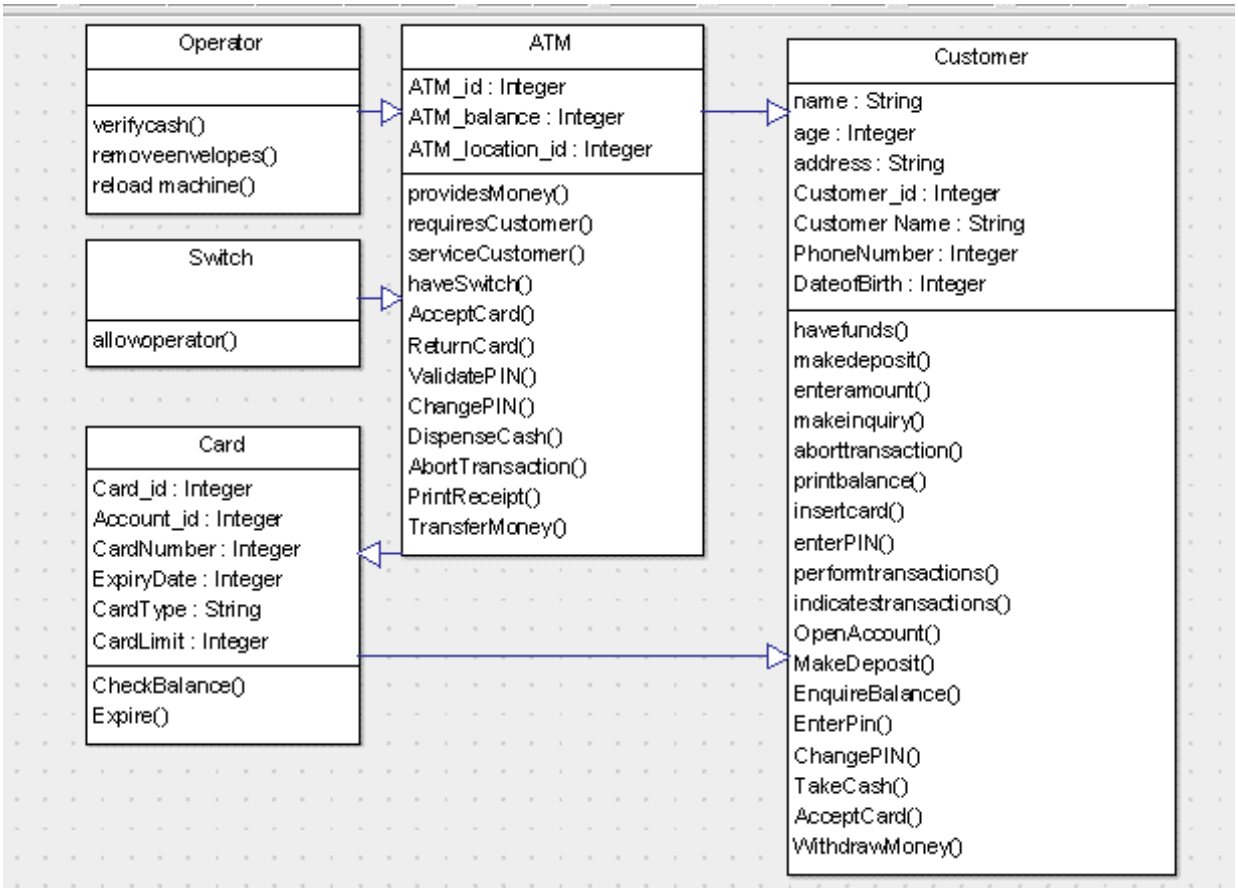


Figure 6.10: Class diagram after enhancement by ontology data

6.8 Abstracting the design with the neural networks

The second part of the learning system is the neural networks and it is used to abstract the design into an architecture for a distributed system. An approach was devised so that the existing capabilities of neural networks could be used to abstract the design for a problem domain. The time series prediction capability of a neural network was used to abstract the design into an architecture. In order to achieve that, the database was once again used. It was decided to use the database to pre-populate data with design elements and then add a historical stream of data which would each represent a design element. The design elements were split into two categories, parent design elements and child(ren) design elements. Each child elements must have a parent element, but each parent may not necessarily have a child element associated to it. Firstly, the program

queries the backend for every parent element and loops on that record set in order to find all the children elements of each parent. Each element (whether parent or child) has a series of values associated with it and these values are fed as input to a time series prediction algorithm. There would be an output for each element and that output is then compared to a threshold value. In case the predicted value exceeds the threshold, that element is included as part of the output of the neural network. In case a parent element is ignored from the neural network output, all the children elements of that element would by default be excluded from the output of the neural network.

6.8.1 Abstraction of the design

This section covers the abstraction of the design. The framework is also set up to refine the design processed by the NLP component and the design extraction component. The text uploaded by the user is processed and a design is generated. In practical terms, the design can be considered as a low-level design as it is derived directly from the text. In a real-world scenario, when an architect is creating the design for a piece of software, the design would be refined to show multiple layers or levels of the software. This would ensure that the design is more modern, splits the tasks by tiers and makes use of the data layer, the service layer and the presentation layer. Inspired by this, the framework has also been enhanced to have a capacity to handle this abstraction of the design, obtained from the text to a more refined design which is more adapted to modern architectures.

In order to achieve this, the database of the framework was extended so that it could be used to refine the design, along with the tables used for the neural networks. It was decided to have three layers of abstraction which would mean that there are at least three levels of granularity of the design. There would usually be four levels of granularity to the design, with the first one being extracted from the text and the next three levels of granular design produced by the framework based on the data available in the database. The three levels of abstraction can be referred to as a fine grain design, a coarse grain design and a thick grain design. The following sections walk through the various abstractions and eventually shows how these levels of granularity are achieved. It would also become clear what exactly is meant by the term “abstraction” and what effect that has on the design.

6.8.1.1Setting up the back-end for abstraction.

The aim of the abstraction is to provide several layers of granular design, using the text input from the user as the starting point. The components described in the previous sections would extract use cases and class diagrams. There are three main database tables used to accomplish the abstraction of the design and they are: bag_of_words_usecase_lowlevel, low_level_transactions, medium_level_transactions and high_level_transactions. Each table would be detailed over the next few sections. The logic applied is that of "bag of words", akin to older natural language processing (NLP) techniques. A set of words, known as a bag of words, would be used to denote a category of behaviour or category of words. The approach adopted uses the database to define the words and bags of words which are subsequently used for abstraction. A "bag of words" or a selection of words is used to denote a low-level concept. Based on the design derived by the previous components, the individual elements are checked against the database, in order to verify if they would abstract to a low-level component. A series of low-level components then abstract to a medium level component and a series of medium level components then abstract to a high-level component. The advantage of such an approach is that it is configurable and the designer can extend the design at all levels of granularity and even for multiple domains. The disadvantage is that the abstraction depends heavily on the content of the database and if the database is not set up to allow for abstraction, the design obtained would be directly from the text.

The first table is the "bag_of_words_usecase_lowlevel" which stores a series of words and what low level concept a particular set of words map to. The low-level transactions are stored in the table "low_level_transactions" and the records in this table are linked to the records in the "bag_of_words_usecase_lowlevel" as well as the table storing medium level transactions, "medium_level_transactions". The table storing the medium level transactions are also linked to the table storing the high-level transactions, "high_level_transactions". The schema diagram below shows the relationships between all these four tables.

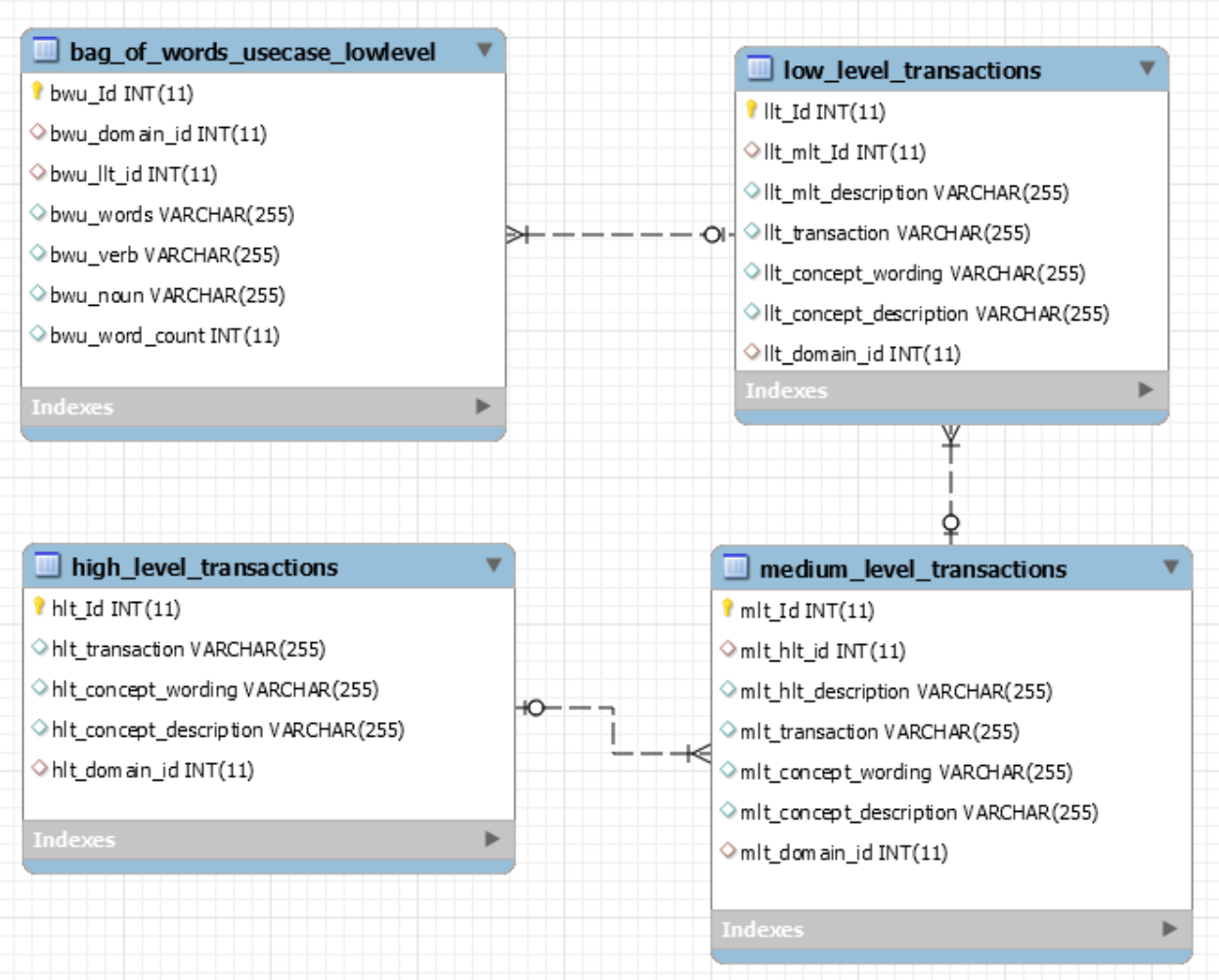


Figure 6.11: Design abstraction table schema

The above-mentioned tables were populated with data pertaining to the ATM domain and the bag of words table was also populated with text or keywords which are relevant to the text used in the case study. When the data from all the tables are extracted and pieced together, the abstracted design would look like as follows:

INPUT	LOW LEVEL	MEDIUM LEVEL	HIGH LEVEL	
	UI	User Interface	ATM User Interface	Front End
Method : insert card Method : enter PIN	Authenticate	Authentication		
Method : make inquiry Method : print balance	Balance (Enquire)			
Method : provides money Method : abort transaction Method : make withdrawal Method : enter amount	Withdraw			
Method : make deposit Method : deposit money	Deposit			
Method : requires Customer Method : service customer Method : perform transactions Method : indicates transactions	Miscellaneous Transactions	Transactions		
Method : verify cash Method : remove envelopes Method : reload machine	Maintenance	Utilities	ATM Logic (transactions etc…)	Service Layer
	Back-end	Storage	Bank Data Store	Back-End

Table 6.17: Abstracted design for case study

6.9 Framework output

After the results have been produced by the framework, the design and architecture can be used to architect a solution for the problem domain specified in the requirements. In order to ensure that the design and architecture are interpreted accordingly, this section explains how the output of the framework is to be understood and used to finalise a design and an architecture.

6.9.1 Processing time

The framework was run multiple in order to gauge the processing time. The following table contains a summary of ten times the framework was executed. The processing time varies according to the processing power and if there are multiple resource intensive applications also running on the framework. For the 10 runs monitored, the average processing time was 2 min 49 seconds.

	Running Time
Run 1	2 min 16
Run 2	4 min 16
Run 3	6 min 46
Run 4	4 min 37
Run 5	1 min 53
Run 6	1 min 39
Run 7	1 min 40
Run 8	1 min 45
Run 9	1 min 43
Run 10	1 min 46
Average Running Time	2 min 49

Table 6.18: Processing time for the framework to run (10 runs)

6.9.2 Interpretation of results

The output of the framework is briefly explained in this section. Firstly, the use cases are produced and they represent the key interactions between the main stakeholders and the system, as well as listing the intended operations. The output of the framework is created by analysing the text. Therefore, some sections of the use cases may seem to be bodily lifted from the text. This can be an advantage as the information in the text is captured in the use cases, even if it can be a little wordy. The four actors identified were *ATM*, *Customer*, *switch* and *operator*. The use case from the framework is evaluated in the future sections by comparing it with the use cases from a manual approach. The second element of design produced is the class diagram. The class diagram is first produced from elements of the text and is then further enhanced by the data in the ontology. The enhanced class diagram is the output of the framework and will later be compared with the class diagram generated by a manual approach.

Thirdly the framework produces a granular abstracted architecture. The granular abstracted design was created as part of the output of the framework as they are closer aligned to the principles of distributed systems. By abstracting the design and considering various levels of granularity for the design, the resulting architecture is more in line with distributed systems, such as SOA (Service Oriented Architecture) principles and micro-service architecture. Thus, in the design of the services and their architecture, the different granularity to be evaluated are fine grain, coarse grain and thick grain.

6.9.2.1 Fine grain

Table 6.17 shows the abstracted design at four different levels. The left most column contains the methods pulled from the text using the NLP component and the design abstraction component. After that a "bag of words" approach was used, with the help of the backend and the neural network, an abstracted design is produced, under the column label "low level". "Low level" stands for low level design, which corresponds to the fine grain architecture. For reminders, the following rows were identified as part of the low-level design.

UI
Authenticate
Balance (Enquire)
Withdraw
Deposit
Miscellaneous Transactions
Maintenance
Back-end

Table 6.18: Low level services for case study

The identified operations and service candidates are mapped to a micro-service configuration and each operation such as Authentication, Balance Inquiry, Withdraw, Deposit, etc, would translate to a micro-service. Each of the fine grain services is to be designed to reflect the business logics and rules. In summary all functional requirements and all operations are considered as services i.e. the ATM application is made up of all the individual services. Clearly this is a fine grain approach, akin to micro-services. Figure 6.12 shows how these fine grain operations can be translated to micro-services in the fine grain approach.

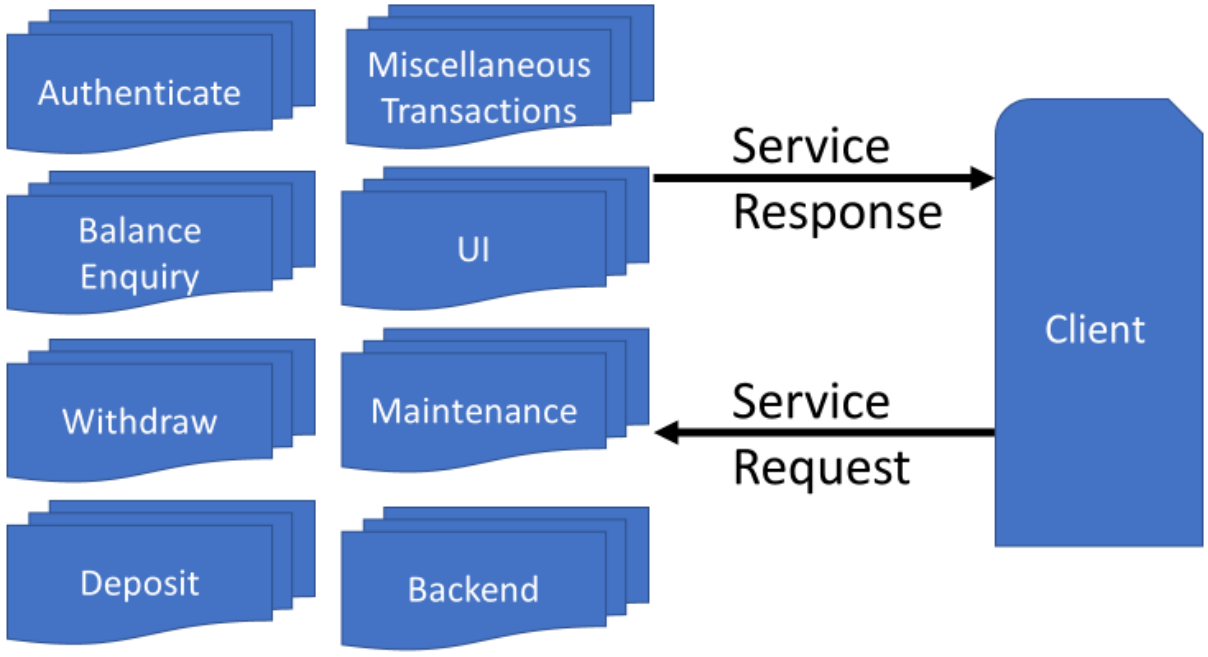


Figure 6.12: Fine grain services

6.9.2.2 Coarse grain

Next some of the operations discussed in the previous sections are aggregated in a logical and consistent fashion to create the ATM application. The aim is to create coarse grain services that are still meeting all the functional requirements. For example, the Transaction Service will comprise of a number of operations such as Withdrawal, Deposit, Balance query and Miscellaneous Transactions, while Authentication will have check Id, Check PIN, change PIN etc. As shown in Figure 6.13 the operations are represented by various components/services that follow the principles of SOA.

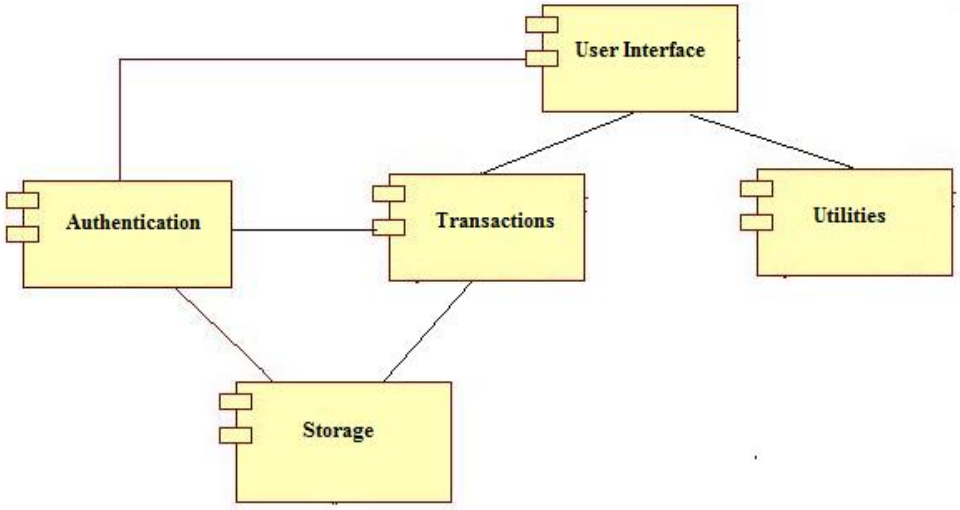


Figure 6.13: Coarse grain services

6.9.2.3 Thick grain

Finally, some of the services/components presented in the previous section are aggregated from the three-tier architecture of the system, with frontend, middleware and backend components, with services/operations mapped to different components still in a logical and consistent fashion to create the ATM application. The architecture is a three-tier architecture, with a backend database on one tier to store all the data, the logical transactions (or service layer) on the middle tier and the final user interface on the last tier. The aim is to create thick grain services that are still meeting all the functional requirements as shown in Figure 6.14.

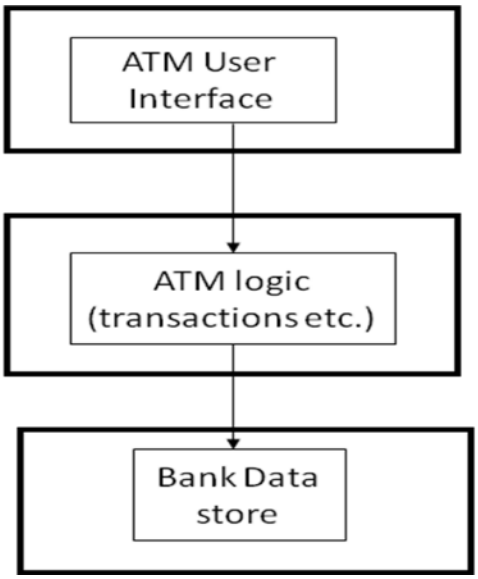


Figure 6.14: Three tier architecture

6.10 Design from a manual approach

In order to evaluate the output of the framework it needs to be compared to the design that has been created manually. The design from a manual approach was created after analysing the requirements and other case studies online which deal with the ATM system. The manual ATM system has been designed in such a way that it is comparable to the output of the framework. It includes a layered or abstracted design which can be used for a distributed system as well as including a class diagram and a series of use cases. The next paragraphs provide a walkthrough of how the requirements were analysed and a design was created for that system manually.

If the design of the ATM system would be expressed as a three-tier architecture, it would consist of a backend tier, a middle tier for all the services and transactions and a user interface tier, which would be ATM terminal where the user interacts with the machine. This three-tier architecture is also often implemented as a three-layer architecture consisting of a data layer, a service layer and a presentation layer. In other web-based three-layer architectures, the presentation layer and the service can sometimes be on the same tier. However, given the nature of the ATM system and the way the customer would use, the three layers are to be distributed on a tier each. Figure 6.15 demonstrates how that architecture would be represented.

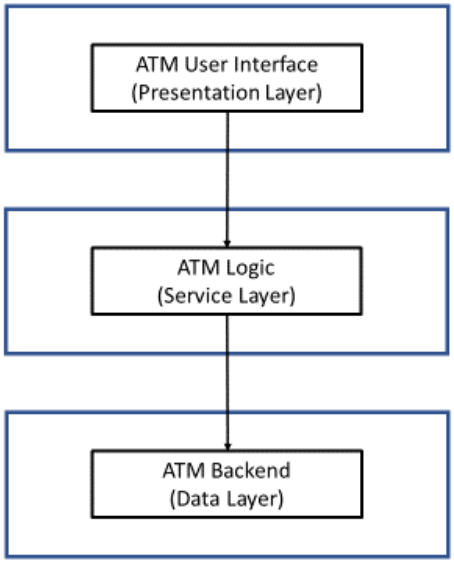


Figure 6.15: Three-tier / Three-layer architecture

The ATM works in conjunction with other banking facilities as almost all the ATM withdrawals and cash payment require the user's data to be updated in real life. The data needs to be able to

synchronize with the bank records so that the user's account is debited or credited accordingly and the user can access the funds they are entitled to when needed. Therefore, the ATM system and the banking system use a series of services to send and synchronize the data. At a high level the ATM machine is based on four main services, Authentication Service for user authentication (including card verification). The Transaction Service reflects the required transactions, withdraw, deposit etc. The Storage Service is used for storing the transactions as well as user details, and Client Service which provides the interface to user of the ATM such as menu and the graphical user interface (Figure 6.16).

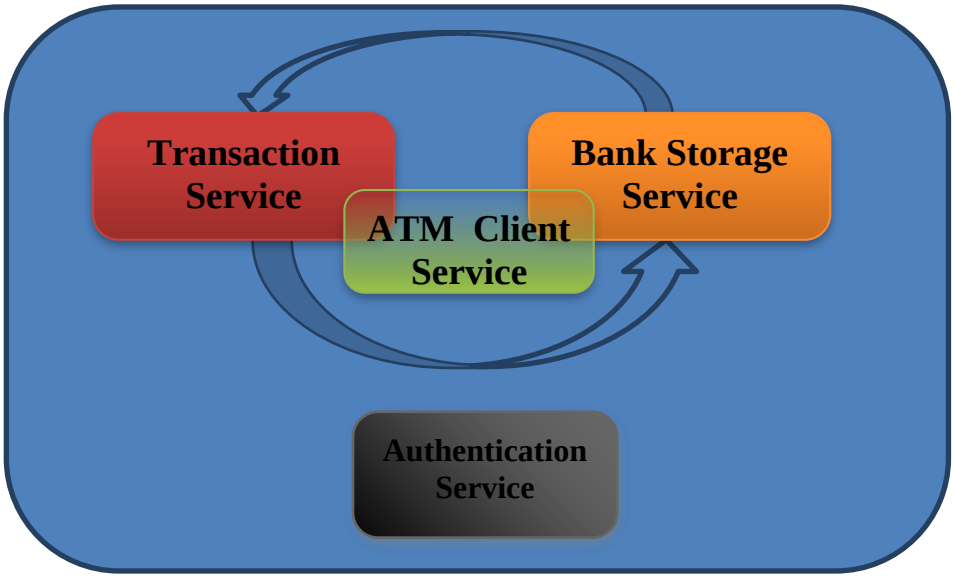


Figure 6.16: ATM system's high-level services

Services are linked together for example the Client Service provides an interface on the local machine to invoke other services such as Authentication and Transaction Services. The same applies to other services for example Authentication Service invokes a signal to other services such as Storage Service checking and verifying users. A number of services, represent the high-level use cases, in the use case diagram (Figure 6.17).

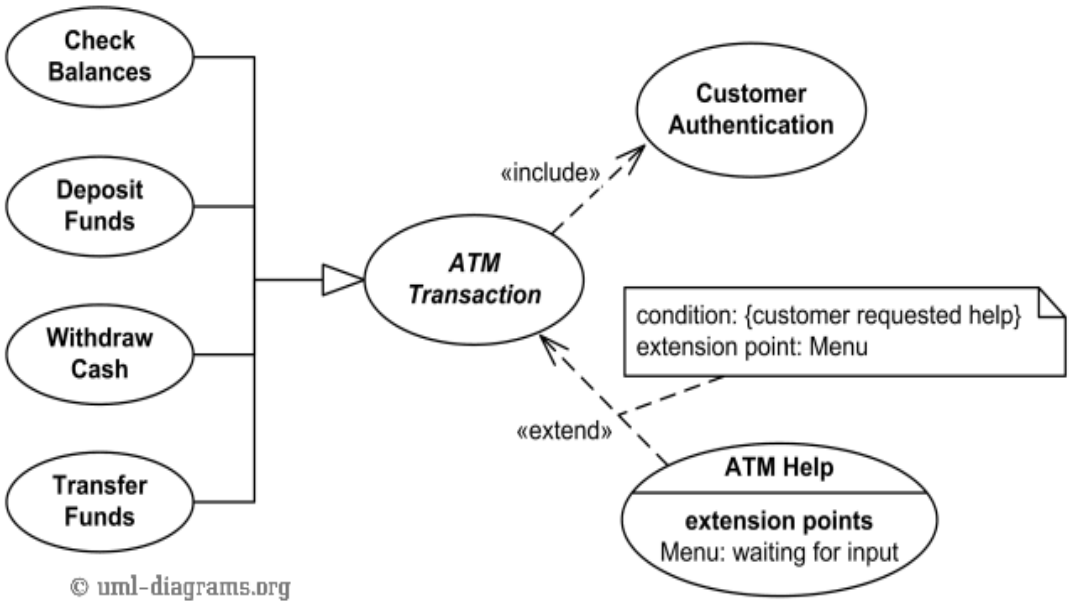


Figure 6.17: Use Case diagram [168]

The use cases representing a low-level design, created from the manual approach, which describes the ATM system acurately and corresponds to the requirement that were described earlier, is shown in Figure 6.18:

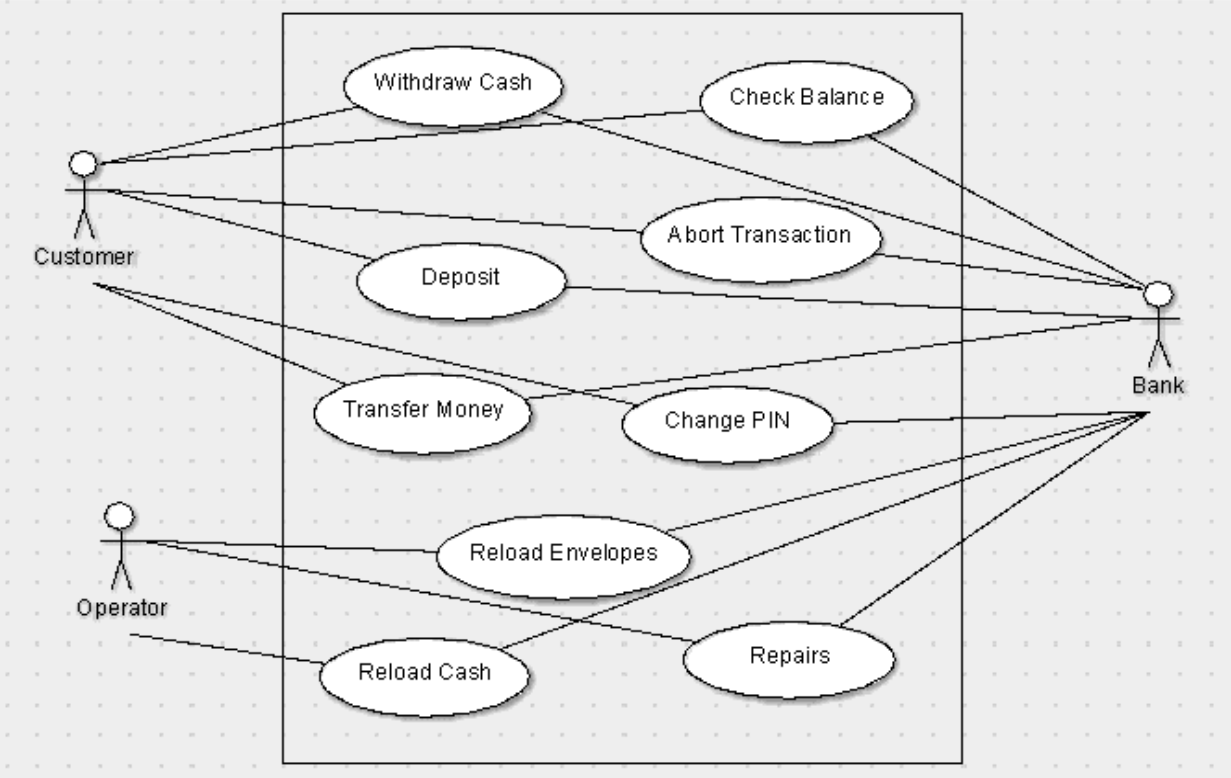


Figure 6.18: Low-level Use Case diagram for an ATM system

A transaction service (Figure 6.19) shows a description of using an ATM machine to withdraw money from a bank account as follows:

- **Insert Card**: The use case begins when the customer inserts their bank card into the card reader of the ATM. The system allocates an ATM session identifier to enable errors to be tracked and synchronized between the ATM and the Bank System.
- **Read Card:** The system reads the bank card information from the card.
- **Authenticate Customer:** Perform Subflow Authenticate customer to authenticate the use of the bank card by the individual using the machine.
- **Select Withdrawal:** The system displays the service options that are currently available on the machine. The customer selects to withdraw cash.
- **Select Amount:** The system prompts for the amount to be withdrawn by displaying the list of standard withdrawal amounts. The customer selects an amount to be withdrawn.
- **Confirm Withdrawal:** Perform Subflow Assess Funds on Hand. Perform Subflow Conduct Withdrawal.
- **Eject Card:** The system ejects the customer's bank card. The customer takes the bank card from the machine.
- **Dispense Cash**: The system dispenses the requested amount of cash to the customer. The system records a transaction log entry for the withdrawal.

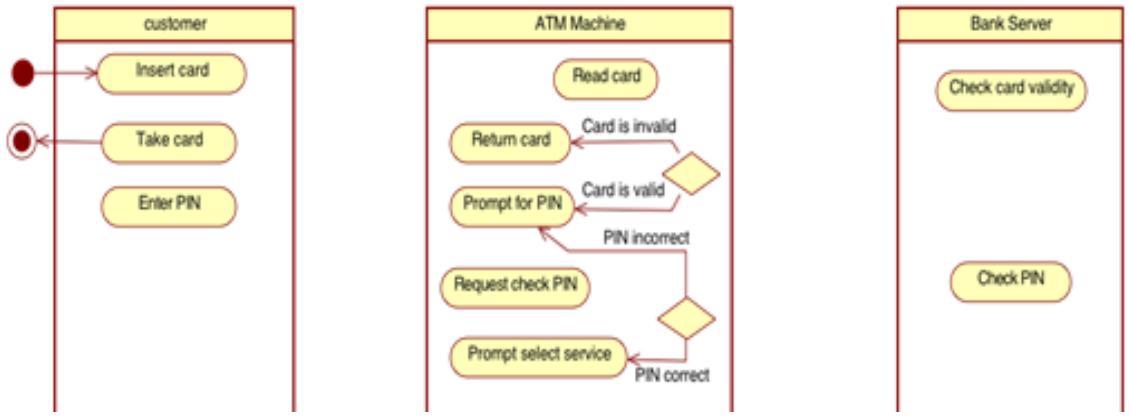


Figure 6.19: Transaction service

When creating the class diagram, several examples were analysed from the web. However, when designing the class diagram, it was important to ensure that the class diagram is relevant to the text (input file) that was used for the case study. For this reason, a custom class diagram was created, with some of the common knowledge of how the ATM system and retail banking works, but by considering what was explicitly mentioned in the text. Therefore, the class diagram from a manual approach is shown in Figure 6.20.

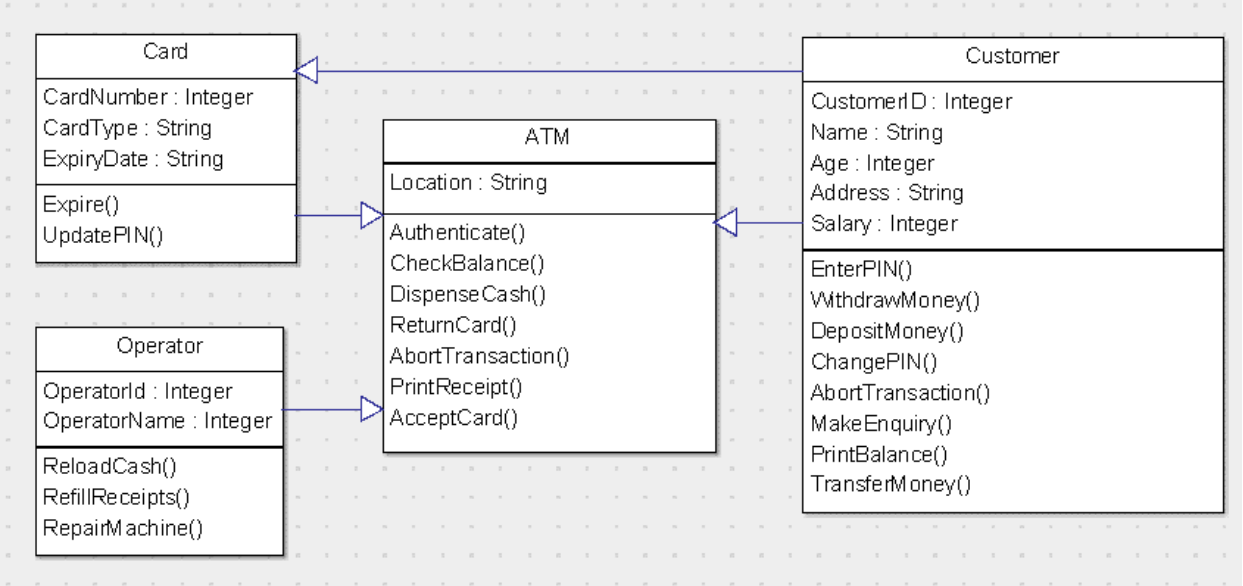


Figure 6.20: Class diagram from a manual approach

Having reviewed how the ATM system usually works and how others have previously designed a system for the ATM system, it is now necessary to evaluate the design obtained and how that could be compared to other designs.

6.11 Evaluation of framework output

The next step is to evaluate the output of the framework and the output can be split into two categories: the design (use cases and the class diagram) and the architecture (the abstracted design consisting of a fine grain, coarse grain and thick grain architecture). The framework is evaluated by comparing the design from the framework to the design obtained from a manual approach. The next section covers the comparison of the design between the manual approach and the output from the framework.

6.11.1 Evaluation of design (use cases and class diagram)

After the framework has processed the input text and produced an output, it is important to analyse the results so as to evaluate the accuracy of the output and assess how the framework is able to generate a design. The output of the framework is analysed so that the meaning of the results produced can be explained and interpreted as a design and an architecture. The problem domain is the ATM system and the requirements describing how the ATM machine should work and interact with the various other systems, all key stakeholders and the bank's database and storage system.

Firstly, the use cases are compared. The manually drawn use cases identify three actors and they are *bank, customer* and *operator*. The use case generated by the framework identifies four actors and they are *ATM, customer, switch* and *operator*. The manual design does not identify ATM as an actor but rather as the system where all the transactions occur. The design generated by the framework have identified four use cases for the ATM and they can be summarised as follows:

- Provides money to authorised customer who have sufficient funds on deposit.
- Requires customer to provide a PIN as an authorisation.
- Service one customer at a time.
- Has a key-operated switch that will allow an operator to start and stop the servicing of a customer.

The first use case where the ATM is described with the capacity to dispense money can be considered as a worthwhile use case while the other three use cases describe features which are important but not critical to the system. The first use case of dispensing money would be more relevant in a class diagram rather than a use case. Therefore, the ATM itself is not an important actor but rather is the system where all the transactions occur. For this reason, it is considered that the ATM is not needed as an actor and can be ignored from the design. It is deemed an irrelevant actor identified by the framework and is not used in the comparison of the design.

The other actors identified by the framework are *customer, switch* and *operator* and the other actors identified by the manual approach are *bank* and *operator*. The actor *switch* should not really be part of the output as it does not describe how the system works but since it passes all the

validation rules, it is present in the output. The following table and graphs describe how the use cases from the manual approach and that generated by the framework compare.

Actors in manual approach		Actors in design generated by framework		Analysis of actors by framework		
Name	**Number of Actors**	**Name**	**Number of Actors**	**Relevant Actors**	**Irrelevant Actors**	**Missed Actors**
Bank		ATM				
Customer		Customer				
Operator	3	Switch				
		Operator	4	2	2	1

Table 6.19: Comparison summary for actors in use cases

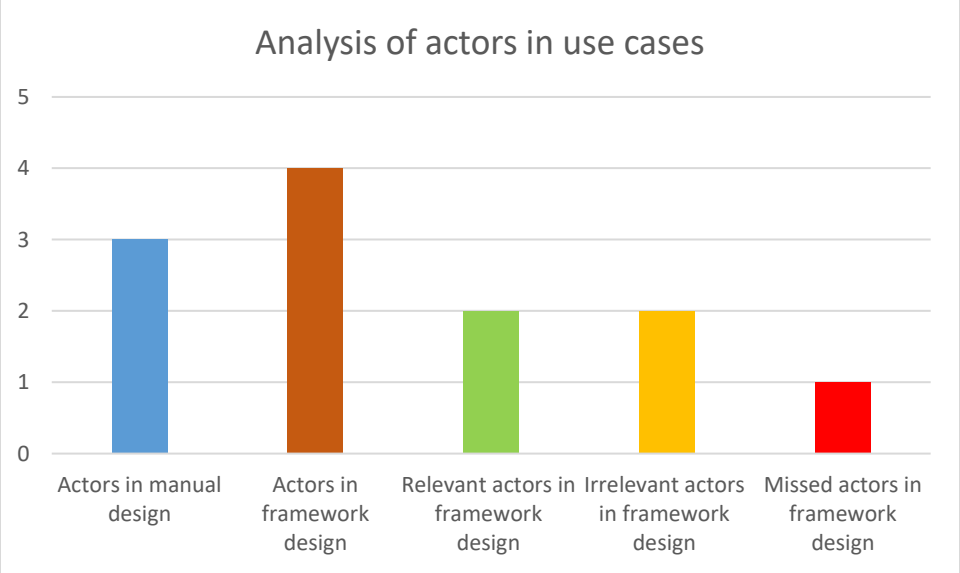


Figure 6.21: Comparison summary for actors in use cases

Next, it is important to analyse the use cases and how the framework fairs in that respect. In this case, the use cases from the actors *ATM* and *switch* would be ignored as they are deemed as irrelevant actors. Before proceeding the diagram below shows all the use cases from the *customer* and the *operator* actors.

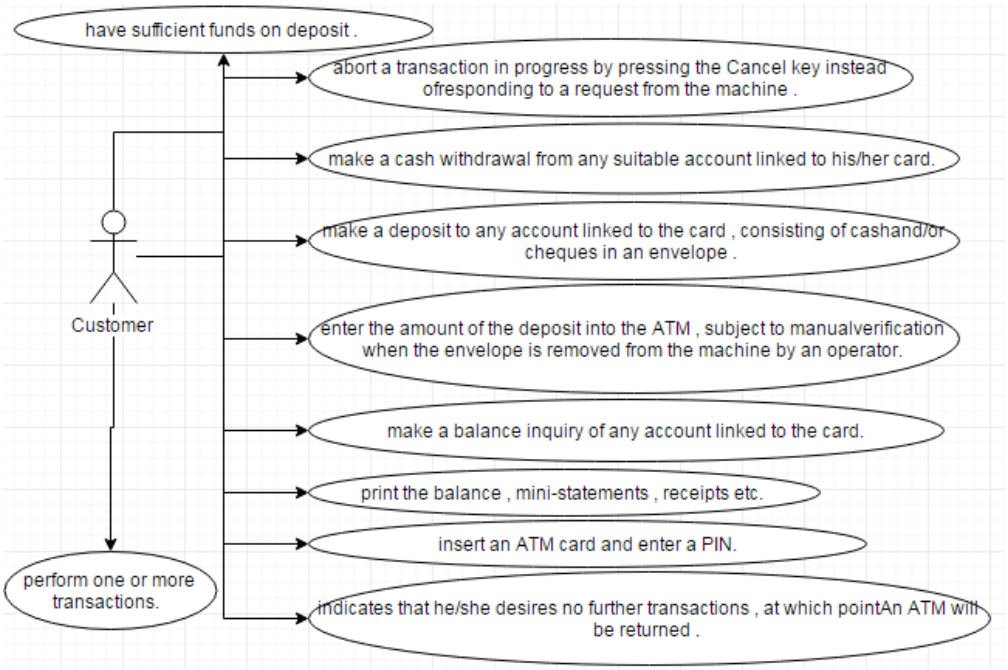


Figure 6.22: Use case for actor *"Customer"* by framework

The use cases contain a lot of information as it is coming straight from the text. Since it is a bit wordy, it shall be simplified so that it can be compared to the design from a manual approach. Furthermore, there are some use cases which are irrelevant to the real purpose and use of the system. Whilst this information is present in the text, they are superfluous or extraneous and have been highlighted in red.

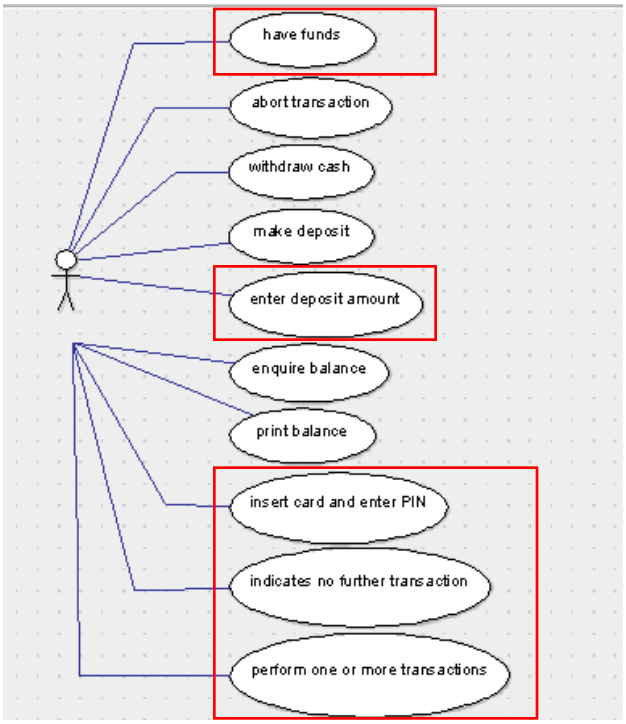


Figure 6.23: Simplified use case for actor *"Customer"* by framework

Therefore, the relevant use cases are: *abort transaction, withdraw cash, make deposit, enquire balance, print balance.* The irrelevant transactions are: "*have funds", "enter deposit amount", "insert card and enter PIN", "indicates no further transactions"* and *"perform one or more transactions".* The actions of the operator can be summarised as "*verify amount of cash"*, which is in line with maintenance and repairs and the other use case would be "*remove envelopes and reload machine"* which is also in line with the transactions of the manual design and compare to the "*reload machine"* use case. The following table and graphs provide a summary of how the use cases can compare. Please note that the table does not contain the data for the actors "*ATM*" and "*switch*" as they are considered as irrelevant actors and do not influence the use cases.

Use Cases in manual approach		Use Cases in design generated by framework		Analysis of Use cases by framework		
Name	**Number of use cases**	**Name**	**Number of Use cases**	**Relevant Use cases**	**Irrelevant Use Cases**	**Missed Use Cases**
Withdraw Cash		Have funds				
Check Balance		Abort Transaction				
Deposit		Withdraw Money				
Abort Transaction		Make Deposit				
Transfer Money		Enter Amount of Deposit				
Change PIN		Enquire Balance				
Reload Envelopes		Print balance & mini statements				
Reload Cash		insert card and enter PIN				
Repairs	9	indicates no further transactions				
		perform one or more transactions.				
		Verify Cash (maintenance)				
		Remove envelopes and reload cash	12	7	5	2

Table 6.20: Comparison summary data for use cases

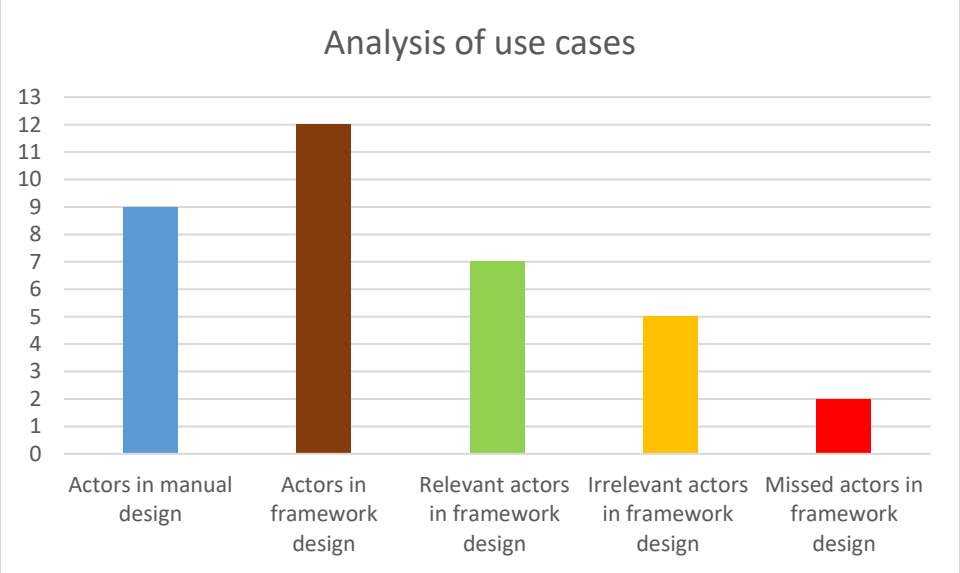


Figure 6.24: Comparison summary data for use cases

The second part of the design consists of the class diagram. The class diagram is an important part of the design as it has its roots from object-oriented programming (OOP) and has been used to design software system. The class diagrams produced by the framework also need to be compared

with the class diagram from the manual approach. Table 6.20 provides a recapitulation of the data obtained from the two class diagrams.

Classes in manual approach		Classes in design generated by framework		Analysis of classes by framework			
Name	**Number of Classes**	**Name**	**Number of Classes**	**Relevant Classes**	**Irrelevant Classes**	**Missed Classes**	**Classes added by ontology**
Card		ATM					
Customer		Customer					
ATM		Switch					
Operator		Card					
	4	Operator	5	4	1	0	1

Table 6.21: Comparison summary for classes in class diagram

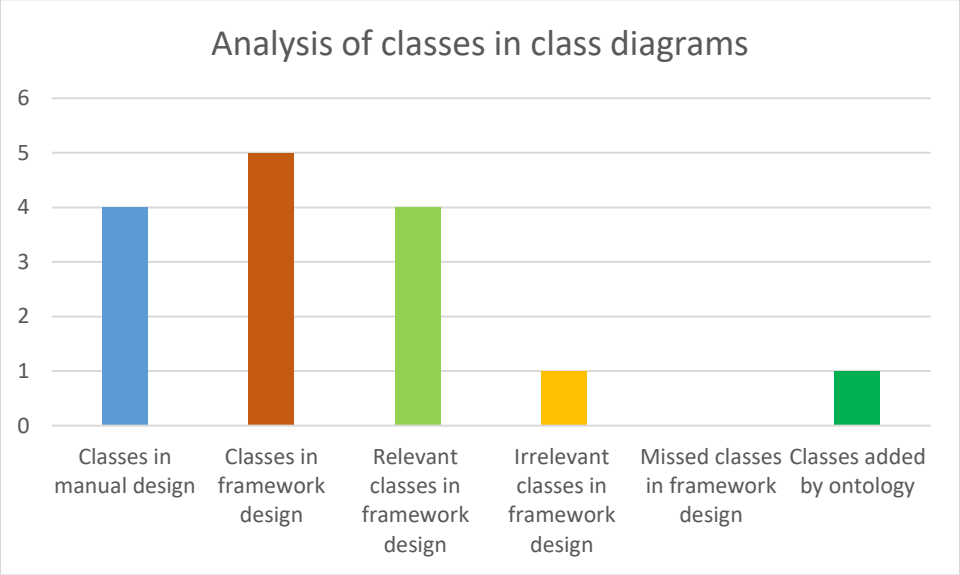


Figure 6.25: Comparison summary for classes in class diagram

After the classes have been compared, the next step is to compare the methods in the class diagram. For each given class, there are a series of methods and attributes. The details listed in the requirements do not provide a lot of information to derive the attributes. Therefore, the analysis focuses on the methods, the number of methods created for each class by the manual approach and the framework, as well as the relevant methods and irrelevant methods listed by the framework and the methods missed by the framework.

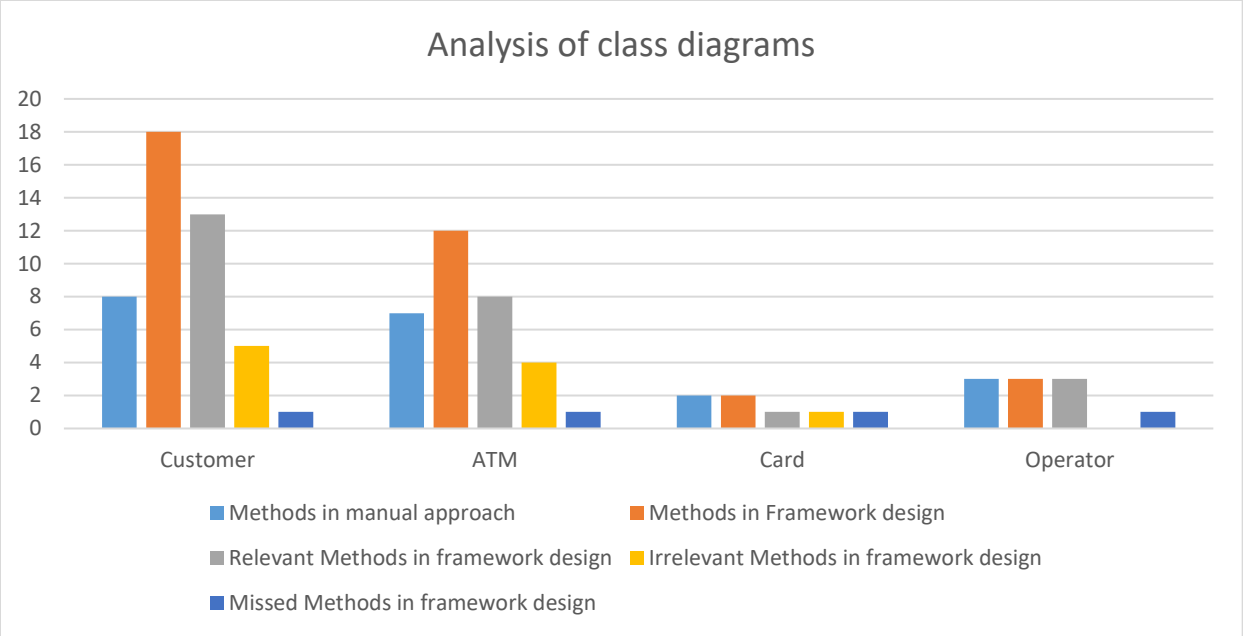


Figure 6.26: Comparison summary for methods in class diagram

Some of the knowledge used to derive the design from the manual approach are inferred from general knowledge and added to the text without being explicitly mentioned in the text. For example, the attribute of the ATM class is "location" which is meant to describe where the ATM is located. Another example is an attribute of the Card class, which is the card number. It is not explicitly mentioned in the text, but it is inferred since a bank card must have a card number.

Regarding the use case diagrams, the manual approach identifies three actors which are *"Customer"*, *"Bank"* and *"Operator"* and the framework identifies four actors which are *"Customer"*, *"ATM"*, *"Switch"* and *"Operator"*. Both approaches identify *"Customer"* and *"Operator"* as actors. "Bank" is identified as an actor by the system designer as it is inferred from general knowledge and it is known that the bank would be carrying out the transactions, but that is not explicitly mentioned in the text. Therefore, the actor *"Bank"* is not identified by the framework. The actor "Switch" is irrelevant and should not be part of the system and the actor *"ATM"* is more considered as the system where transactions occur rather than an actor. Out of the five actors identified by both approaches, *"Customer"* and *"Operator"* are common.

Moreover, for the use cases, the manual approach identifies nine key transactions which are "Withdraw Cash", "Check Balance", "Deposit", "Abort Transaction", "Transfer Money", "Change PIN", "Reload Envelopes", "Reload Cash" and "Repairs". Out of these nine transactions,

the framework identifies six (“Withdraw Cash”, “Make Deposit”, “Enquire Balance”, “Abort Transaction”, “Reload Envelopes” and “Reload Cash”). Therefore 67% of the key use cases (6 out of 9) are identified by the framework.

Regarding the class diagrams, the manual approach identifies four classes and they are “Card”, “ATM”, “Operator” and “Customer”. The framework identifies all four of these classes and also identifies a fifth class, “switch”, which is not relevant to the system. The framework is able to identify 100% of the relevant classes (4 out of 4), even if the class “card” is added from the ontology data and is not picked up from the text.

Regarding the attributes, the manual approach identifies two attributes (name and id) for the “Operator” class, and as this is inferred from general knowledge and the framework does not identify any attributes. Both approaches identify the three methods, for the class “Operator”, for the maintenance of the cash machine. For the “ATM” class, the manual approach infers location as an attribute and the framework lists (ATM_Id, ATM_Balance, ATM_Location_id) which is added through the ontology. For the “Customer” class, the manual approach identifies five attributes (CustomerId, Name, Age, Address and Salary) and the framework identifies five attributes (name, age and address from the text and Customer ID, Phone Number and Date of birth from the ontology). The framework misses the attribute “salary” from the text. The manual approach identifies eight methods (Enter PIN, Withdraw Money, Deposit Money, Change PIN, Abort Transaction, Make Enquiry, Print Balance, Transfer Money) and the framework identifies 18 methods. There are four irrelevant methods (Have Funds, Enter Amount, Perform Transactions, Indicate Transactions) and six relevant methods (Make Deposit, Make Enquiry, Abort Transaction, Print Balance, Insert Card and Enter PIN) identified from the text. Finally, there are eight methods added from the ontology (Open Account, Make Deposit, Enquire Balance, Enter PIN, Change PIN, Take Cash, Accept Card and Withdraw Money). There are two duplicate methods which are identified both in the text and added by the ontology, and they are Make Deposit and Enter PIN.

6.11.2 Evaluation of architecture (abstracted design)

The framework also produces an architecture, which has been referred to as the “abstracted” design and which is more suitable for distributed systems. A distributed architecture was also created manually and was described in section 6.8. The considerations for an architecture done manually was influenced by common knowledge such as the fact that an ATM for a bank may use services

from other banks to allow cash withdrawal. These pieces of information are not explicitly mentioned in the text but are used to create a distributed architecture for the ATM system.

Firstly, the most abstracted designs (thick grain designs) are compared. Both designs propose a typical three-tier architecture and Figure 6.27 shows both architecture side by side. There is provision for a user interface tier, a service-tier and a backend-tier in both cases. The architecture would be very similar if not the same as the high-level view of a three-tier architecture provide details of only the tiers to be created.

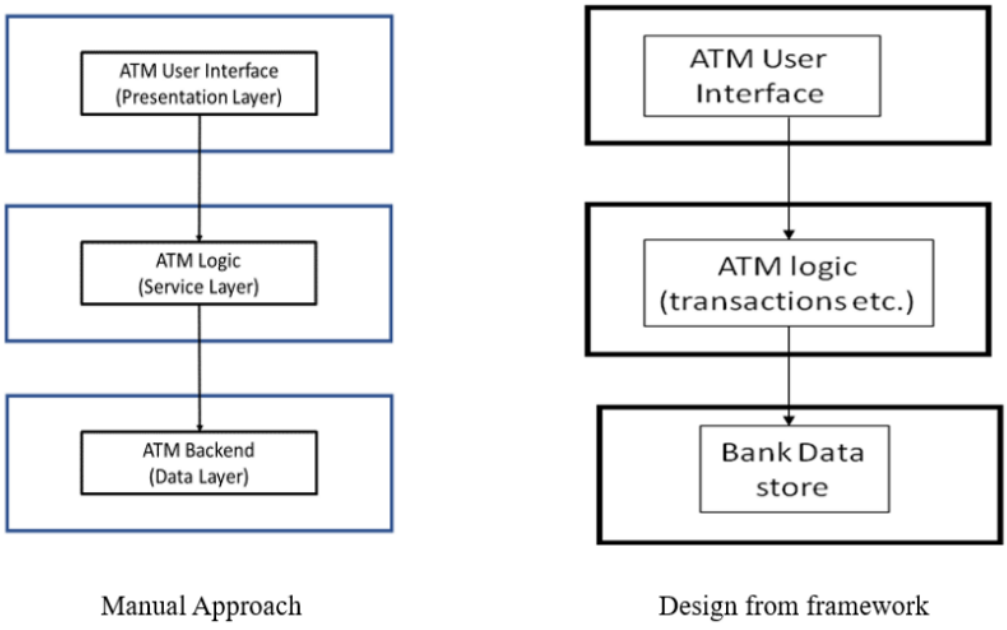


Figure 6.27: Comparison of tier architectures (thick grain) from manual approach and framework

Next the design which is one grain finer, the coarse grain architectures are compared. The framework produced a coarse grain architecture by using the knowledge base in the backend and the neural network to create a coarse grain architecture from the fine grain architecture. For the manual approach, the need for services and the knowledge of the industry, such as the fact that there would inter-bank services to allow withdrawal from other banks ATM's. Figure 6.28 shows both coarse grain architecture side by side.

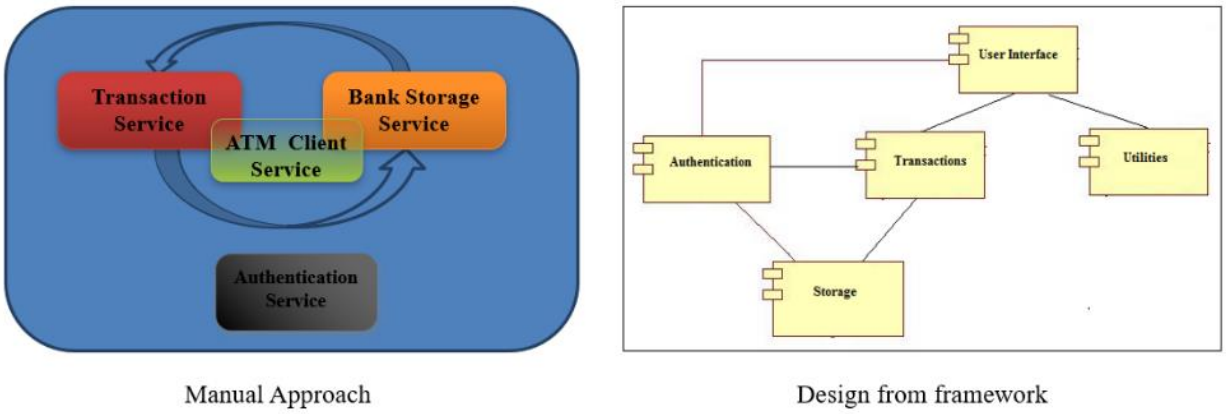


Figure 6.28: Comparison of tier architectures (coarse grain) from manual approach and framework

There are four services in the manual design and five services in the output from the framework. The framework includes the *"Utilities"* service whereas the manual approach does not. The *"Utilities"* service is added through the abstraction process. The data present in the backend is processed and the neural network picks up the services which have a computed score exceeding a threshold value. The other four services are equivalent: Transaction Service maps to Transactions, ATM Client Service maps to User Interface, Bank Storage service maps to Storage and Authentication Service maps to Authentication. In this case it can be considered that the framework is more accurate as the description of the Utilities service would allow miscellaneous transactions to be coded, without the need to extend on the design. The manual approach provides 80% of the design of ATM system as it contains 4 out of 5 of the services required for the system.

Finally, the fine grain architectures are compared. The framework elaborated a fine grain architecture using a "bag-of-words" approach which uses data in the backend and the neural network to create a fine grain architecture. For the manual approach, the use cases were analysed, along with the sub-flow of cash withdrawal and inferred data on how the ATM system works in general. Previously the micro-service architecture from the manual approach was not shown for the sake of brevity and it is shown below, in comparison with the fine grain architecture from framework.

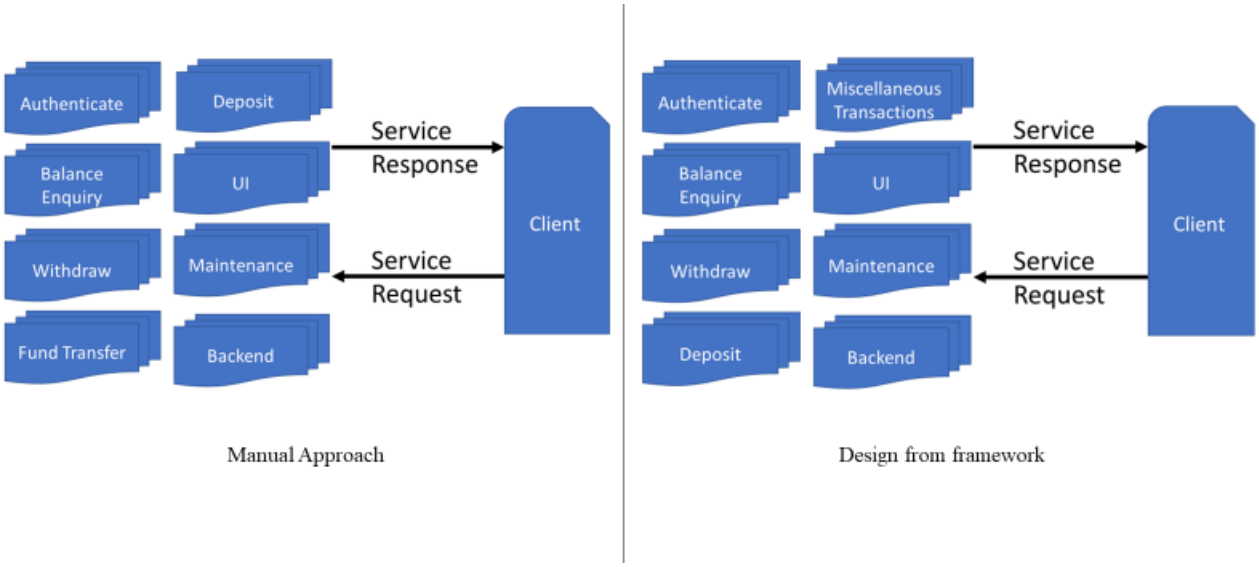


Figure 6.29: Fine Grain Architecture / Micro-services architecture comparison

The following table summarises the data and the graph is plotted using the data from the table.

Micro-services in manual approach		**Micro-services in design generated by framework**		**Analysis of Micro-services by framework**		
Name	**Number of services**	**Name**	**Number of services**	**Relevant Services**	**Irrelevant Services**	**Missed Services**
UI		UI				
Authenticate		Authenticate				
Balance (Enquire)		Balance (Enquire)				
Withdraw		Withdraw				
Fund Transfer		Deposit				
Deposit		Miscellaneous Transactions				
Maintenance		Maintenance				
Back-end	8	Back-end	8	7	1	1

Table 6.22: Comparison summary for services in fine grain architecture

The fine grain architecture from the framework identifies 7 out of the 8 services described in the manual approach (UI, Authenticate, Balance (Enquire), Withdraw, Deposit, Maintenance and Back-end). So, 87.5% of the fine grain architecture from both approaches match. The framework fails to identify “Fund Transfer” and identifies “Miscellaneous Transactions”, which is not mentioned in the manual approach.

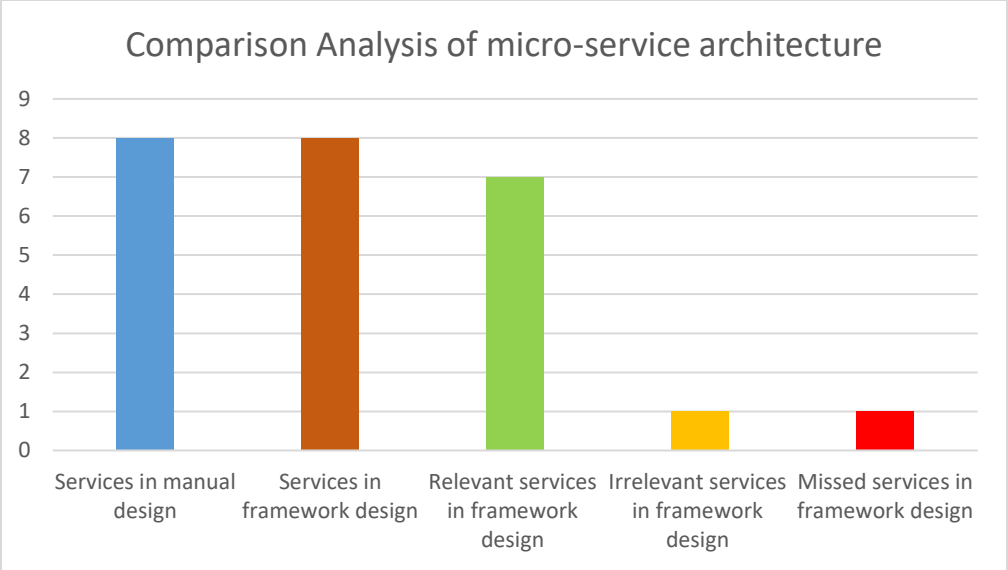


Figure 6.30: Comparison summary for services in fine grain architecture

This wraps up the analysis and comparison between the design and architecture obtained by the framework to the ones drawn manually. The similarities and differences have been explained and both sets of design and architecture describe the possible implementation of the ATM system, with minor differences in the architecture. As explained before, the proposed approach is a semi-automated one and the architect should now consider the output of the framework.

6.12 Conclusion

The goal of the framework is to provide a semi-automated approach to partially automate the requirements into a design and an architecture, which are better suited for distributed systems. To a large extent, the framework is able to deliver on these needs. In so doing, the hope is that the solution produced by the framework is better aligned with the requirements as it sources its input from the text and goes on to propose a design and an architecture. The output of the framework is presented to an architect, who still has the responsibility to finalise the architecture, given the brief of client's requirements, beyond the scope of the requirements mentioned in the text.

The framework sets out to propose an architecture which is distributed, while using a learning system which allows the framework to learn and grow and also reduce the introduction of defects from the requirements into the design phase. The framework is able to deliver largely on these needs, as shown with the evaluation of the case study. The design and architecture produced may contain elements of design which are not desirable, but the user can configure the learning system

to ignore these elements and over time there should be less and less user configuration for a domain. Whilst the system is not perfect and may need a fine tuning, it does resolve the majority of the issues that it sets out to do.

There are also some limitations with the case studies as they may not present complete requirements for an ATM, and they do not describe extensive dependencies and complex requirements of existing systems with which the software would be interacting with. In the industry, requirements might be more complex and technical for maintenance projects. However, when it comes to new projects, which are to be architected from scratch, the framework would be able to provide a high-level architecture, a medium-level architecture and a low-level architecture, provided there are adequate data for the learning systems. While various individual aspects of the framework could be improved, the current version of the framework does a satisfactory job at providing a distributed architecture, coupled with a learning which allows it to grow over time.

In an ideal world, the output from the framework would be a very good match for the problem domain and would be very close to a solution, ready to be handed over to the implementation team. However, when considering an architecture, there are many factors to be considered which are often not explicitly mentioned in the requirements. For example, should the architecture be a distributed one, correspond to traditional Object-Oriented Programming (OOP) or use a monolithic architecture. If the software to be created is going to extend an existing platform, should the architecture of the software be compatible with that of the existing platform. For these reasons the framework can provide an architecture that suits different problem specifications, by proposing three types of architectures.

CHAPTER 7: CONCLUSION

7.1 Summary

Defect detection and fixing is an important part of software development as the goal is to reduce the number of defects and time spent on reworks as well as improving the user experience. However, the majority of software are developed to completion and then go through a quality assurance (QA) process which is aimed at finding and fixing defects. There can be more done to avoid the introduction of defects into the SDLC in the first place, with the goal to reduce the introduction of defects and not wait for post-implementation QA to find and remove defects. Since defects introduced in the earlier phases of the SDLC can be the costliest to fix, focusing on the reduction of defects from the earlier phases of the SDLC, can be very advantageous.

Requirement defects leak into the later phases of the SDLC and can cause defects at all stages. The design can be particularly at risk as the design ensues from the requirements. Poor design and software defects often make source codes hard to understand and lead to maintenance issues. Whereas early detection and fixing of defects make programs easier to understand by developers. Implementation of corrective and preventive actions can improve software quality. The correction solutions should minimize, as much as possible, the number of defects detected during the quality assurance phase. Defect prevention practices enhance the ability of software developers to learn from those errors and learn from the mistakes of others. Effective defect tracking involves a structured problem-solving methodology to identify, analyse and prevent the occurrence of defects. Defect prevention is an ongoing process of collecting the data, performing root cause analysis, determining and implementing the corrective steps and sharing the lessons learned to avoid future defects.

The aim of the proposed framework created is to reduce the introduction of defects from the requirement and design phases. The work carried out throughout the thesis is summarized next. The research began with an overview of requirement defects, including requirement defect identification and classification. Requirement categories and desirable requirement attributes were also described. Manual defect detection techniques, including reading techniques were described

as well as defect prevention activities aimed at the requirement phase. The research also covered automation within the SDLC including Code Generation, Configuration Management, Quality Assurance and Testing (Automation Testing), software building and DevOps. Commercial tools available to manage requirements and textual analysis techniques which have been used to either extract features from the requirements or automatically translate the requirements into a design were also covered. The existing approaches were critically analysed and the need for a new framework was explained.

This research has focused primarily on the development of a framework which can semi-automatically derive an architecture from the natural language requirements, which is better adapted for distributed systems. The design is abstracted so that it can be implemented as a distributed system, and the output of the framework consists of three levels of abstraction, which are fine grain, coarse grain and thick grain architectures. Ideally the fine grain architecture would be implemented as a micro-service architecture, the coarse grain as a Service Oriented Architecture (SOA) and the thick grain as a three-tier architecture. The output of the framework also includes a design which is suitable for OOP implementation and consists of a series of use cases and a class diagram. The framework also makes use of a learning system which consists of an ontology and a neural network. The ontology enhances the design and the neural network is used to abstract the design into an architecture.

The framework was evaluated with the help of a case study and the output from the framework was compared to the output from a manual approach. The results were discussed and the similarities and differences between the two approaches were highlighted. The case study selected was within the financial domain, specifically regarding the design of an ATM system. The framework is able to successfully produce a design, consisting of a class diagram and use cases, as well as an architecture which proposes three levels of granularity. The output of the framework provides several choices for an architecture, which are all based on distributed systems, and the architect can choose the best fit and adapt it for his needs. The process has been semi-automated for the architect.

The novelty of the framework lies in the fact that the design (class diagram and use cases) obtained from the natural language requirements are semi-automatically translated into an architecture for distributed systems. The approach is semi-automatic as there needs to be enough data in the

database to allow the abstraction of the design into an architecture. The existing approaches do not produce an architecture which is suitable for distributed systems. Furthermore, the architecture produced comes in three levels of granularity, which means that the architect is free to choose which type of architecture (micro-service, SOA or three-tier) is best adapted for the needs of the project. The framework also makes use of an ontology data to allow the user to control part of the output. The user can therefore decide which parameters are absolutely required and which are the ones which can be ignored. The learning system also comprises of a neural network which is used in the abstraction of the design. The abstraction of the design depends on the data in the backend and is used to transform the design into an architecture. The fact that a neural network is used also means that the framework is equipped with a learning system, and that the output of the framework is dynamic and the system can be configured to learn and change over time.

7.2 Future work

The case study was able to showcase the ability of the framework and demonstrate how a design and an architecture can be derived. However, it was also noticed that there are certain areas which can be improved. So, based on the work presented in this thesis, there are a number of areas that can be further improved and carried forward.

The work presented so far has been able to demonstrate how the framework can be applied to a domain and generate UML design and an architecture which can be used for distributed systems. The architecture can be applied to either micro-services, service-oriented architecture (SOA) or a three-tier architecture. The case study has demonstrated the potential of the framework, but the applicability of the framework is limited. In order to further evaluate the framework, it should be tested on a larger case study and also multiple case studies within the same domain. This would help to extend the ontology data and the learning system so that they can cover a particular domain more effectively. The framework could also be applied to real life scenarios or scenarios which are within different domains.

The next aspect of the framework which could be improved is the NLP parser. The main parser used is the Stanford parser. There are now different kinds of parsers which use neural networks and derive vectors to represent each word. The framework could be improved using one of these parsers. In fact, half way through the research, the framework was changed to use a vector-based

parser. The aim was to use the distance between words, which are being represented as either 50 or 100 or 300 dimension vectors, to calculate the distance (and hence a coefficient of relatedness) between the words. Vector-based NLP parsers was used to parse the requirements documents but could not be used to extract concepts, a design and an architecture from the text. Given also the time constraint, the parser was reverted back to the original Stanford parser so that the research results could be demonstrated. A future piece of work would include using a vector-based parser as the main parser and then customize the output to derive a design and an architecture.

The next part of work which could be considered is the classification of the requirement documents. This would amount to the automatic detection of domains. During the experiments with the vector-based parser, a document containing 1000 words was used as input and a lookup file containing the 50 dimensions for 100 000 words was used. The input document containing 1000 words was classified into 50 clusters, using K-Means and K-Means Plus algorithm. The result was that in each of the 50 clusters, the words were very related. For example, one cluster would contain many words within the banking sector, another contained words within the aviation industry and so on. Therefore, this could be extended so that the framework can automatically detect domains and use the correct one for a requirement document. This would also mean less manual work needed to set up a domain's data in the database prior to running the software.

The next piece of future work would include changing the learning system, especially the neural network to derive an architecture out of the design. Currently the architecture is expressed and a series of values which are then used by a time series prediction. At this stage, it is believed that time series prediction and a series of numerical values are inevitable to be used by the neural network. However, the way the data is represented can be further researched. A design, including the class diagram and use cases can also be represented as a series of numerical values. A piece of research could focus on formalising the design into a series of objects which each have multiple values to represent a design artefact. The historical values for each element can be kept in a database and then used to predict the design. The same theory could also be applied to the abstraction of the architecture. The three levels of granularity of each architecture could be further broken down in a series of elements each represented by a numerical value. A list of historical values is kept in the backend and used to predict a design. The approach is similar to what is currently being done, but it is proposed to have more elements to represent a design or an

architecture. This approach could also be linked to the vector-based parser and predict an architecture based on the vector values found in the text. This research could amount to another Phd project.

The final piece of enhancement which could be brought to the framework would be a self-learning or changing system to allow the output of the framework to change over time. This could be two-folds. Firstly, the user feedback should be collected in a different manner. When the output of the framework is presented to the user, the latter should be able to make corrections. These corrections should then be automatically registered in the backend and be used for future predictions. In this way, the ontology data would be maintained semi-automatically and not manually. The second aspect would require more research and would also involve user feedback. Whenever the user is presented with an architecture, he should be allowed to alter the architecture (abstracted design). These changes then need to translate into a series of values which update the data for the neural network. Thus, the user feedback is more prominent in the system and the neural network is closer to a self-learning system. Each output computes a series of values which are added to the historical data, which is then later used to predict the abstracted architecture again.

Overall, there are several avenues which can be considered to further continue research on the framework or the same topic. The goal of this research was to prove the concept of generating a design and an architecture for distributed systems, while using a learning system and accepting natural language requirements as input. The research was able to deliver on that aspect.

Before concluding, it would be interesting to outline the aspects of the research work which could be published next. Firstly, a paper explaining the current form of the framework could be considered for publication. The initial papers published described an older version of the framework which has not been implemented in the form that was described in the publications. The output from the case study and some additional information could be mentioned to highlight the performance of the framework. Secondly, a paper could be considered after some additional work has been completed on the framework. When the vector-based parser has been implemented and the framework is able to semi-automatically detect the domain, there would be enough interesting material to consider the publication of a second. To conclude, it would be good to mention that there are different publications opportunities which can be explored.

BIBLIOGRAPHY:

1. Cerpa, N., & Verner, J. M. (2009). Why did your project fail?. Communications of the ACM, 52(12), 130-134.
2. Zhivich, Michael, and Robert K. Cunningham. “The Real Cost of Software Errors.” *IEEE Security & Privacy Magazine* 7.2 (2009): pp 87–90. © 2012 IEEE
3. Charette, Robert N. "Why software fails." *IEEE spectrum 42, no. 9* (2005): 36.
4. “A study in project failure”, http://www.bcs.org/content/conwebdoc/19584 Accessed 24th April, 2014
5. Mäntylä, Mika V. and Juha Itkonen. "How are software defects found? The role of implicit defect detection, individual responsibility, documents, and knowledge." *Information and Software Technology* (2014).
6. Adeel, Kashif, Ahmad Shams and Akhtar Sohaib. "Defect prevention techniques and its usage in requirements gathering-industry practices." *Engineering Sciences and Technology, 2005.* SCONEST 2005. Student Conference on. IEEE, 2005. pp 1 – 5.
7. Boehm, Barry and Victor R. Basili, “Software Defect Reduction Top 10 List,” *Computer*, vol. 34, no. 1, Jan. 2001, pp 135–137
8. “Reducing rework through effective requirements management.” (2009), http://public.dhe.ibm.com/common/ssi/ecm/en/raw14192usen/RAW14192USEN.PDF Accessed 19th June, 2014
9. Buyya, Rajkumar, Chee Shin Yeo, Srikumar Venugopal, James Broberg, Ivona Brandic, “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility”, *Future Generation Computer Systems*, Volume 25, Issue 6, June 2009, pp 599-616, ISSN 0167-739X, http://dx.doi.org/10.1016/j.future.2008.12.001. (http://www.sciencedirect.com/science/article/pii/S0167739X08001957) Accessed 5th October 2013.
10. Weinhardt, Christof, Wirt Arun Anandasivam, Benjamin Blau, Nikolay Borissov, Thomas Meinl, Wirt Wibke Michalk, and Jochen Stößer, (2009). “Cloud computing–a classification, business models, and research directions”. *Business & Information Systems Engineering*, 1(5), pp 391-399.

11. Rings, Thomas, Geoff Caryer, Julian Gallop, Jens Grabowski, Tatiana Kovacikova, Stephan Schulz, Ian Stokes Rees. “Grid and Cloud Computing : Opportunities for Integration with Next generation Network”. *Number 3, s.l J Grid Computing, 2009, Vols. Volume 7*, ISSN 1570-7873 (Print) 1572-9814 (Online)
12. Khaddaj, Souheil, and Gerard Horgan. "The evaluation of software quality factors in very large information systems." *Electronic Journal of Information Systems Evaluation 7.1* (2004): pp 43-48.
13. IEEE, IEEE Standard for Software Reviews, IEEE Std 1028–1997, 1998, pp. i–37.
14. I.L. Margarido, J.P. Faria, R.M. Vidal, M. Vieira, Classification of defect types in requirements specifications: Literature review, proposal and assessment, Presented at the 2011 6th Iberian Conference on Information Systems and Technologies (CISTI), Chaves / Portugal, 2011.
15. G.S. Walia, J.C. Carver, A systematic literature review to identify and classify software requirement errors, Inform. Softw. Technol. 51 (2009) 1087–1109.
16. Latino, R.J., Latino, K.C, & Latino, M.A. (2011). Root Cause Analysis: Improving Performance for Bottom-Line Results. CRC Press.
17. https://www.isixsigma.com/tools-templates/software/defect-prevention-reducing-costs-and-enhancing-quality/, Accessed 11th March 2019.
18. Blackburn, Mark R., Robert Busser, and Aaron Nauman. "Removing Requirement Defects and Automating Test." STAREAST, May (2001).
19. “Software Errors Cost U.S. Economy $59.5 Billion Annually, NIST Assesses Technical Needs of Industry to Improve Software-Testing”, http://www.abeacha.com/NIST_press_release_bugs_cost.htm, http://www.cse.buffalo.edu/~mikeb/Billions.pdf Accessed 30th April, 2014
20. The Defect Prevention Process, https://flylib.com/books/en/1.428.1/the_defect_prevention_process.html, Accessed 10th October, 2019
21. Universidad Politecnica de Valencia. Architecture Evaluation Methods: Introduction to ATAM.

22. Alshazly, A. A., Elfatatry, A. M., & Abougabal, M. S. (2014). Detecting defects in software requirements specification. Alexandria Engineering Journal, 53(3), 513-527.
23. Kamsties, Erik, and B. Peach. "Taming ambiguity in natural language requirements." In Proceedings of the Thirteenth International Conference on Software and Systems Engineering and Applications. 2000.
24. Berry, Daniel M. "Ambiguity in natural language requirements documents." In Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs, pp. 1-7. Springer Berlin Heidelberg, 2008.
25. Wilson, W. M., Rosenberg, L. H., & Hyatt, L. E. (1997, May). Automated analysis of requirement specifications. In ICSE (Vol. 97, pp. 161-171).
26. Stokes, David Alan, Requirements Analysis, Computer Weekly Software Engineer's Reference Book, 1991, pp. 16/3-16/21.
27. IEEE Std 830-1993, Recommended Practice for Software Requirements Specifications, December 2, 1993.
28. Curtis, B. (1981). The measurement of software quality and complexity. Software Metrics: An Analysis and Evaluation, 203-224.
29. Sommerville, Ian, Software Engineering, Fourth Edition, Addison-Wesley Publishing Company, Wokingham, England, 1992.
30. Clements, P., Kazman, R., & Klein, M. (2003). Evaluating software architectures. Beijing: Tsinghua University Press.
31. Hammer, T. F., Huffman, L. L., Rosenberg, L. H., Wilson, W., & Hyatt, L. E. (1998). Doing requirements right the first time. CROSSTALK The Journal of Defense Software Engineering, 20-25.
32. Kott, A., & Peasant, J. L. (1995). Representation and management of requirements: The RAPID-WS project. Concurrent Engineering, 3(2), 93-106.
33. M. E. Fagan, "Design and Code Inspection to Reduce Errors in Program Development," IBM Systems Journal, vol. 15, 1976, pp. 182-211
34. Oliver Laitenberger, "A Survey of Software Inspection Technologies", Handbook on Software Engineering and Knowledge Engineering, Citeseer, 2002.
35. Sulehri, L. (2010). Comparative Selection of Requirements Validation Techniques Based on Industrial Survey.

36. G. Sabaliauskaite, F. Matsukawa, S. Kusumoto, K. Inoue, An Experimental Comparison of Checklist-Based Reading and Perspective-Based Reading for UML Design Document Inspection, Presented at the Proceedings of the 2002 International Symposium on Empirical Software Engineering, 2002.
37. F. Lanubile, G. Visaggio, Evaluating defect detection techniques for software requirements inspections, International Software Engineering Research Network (ISERN) Report no00-08, 2000.
38. M. Ciolkowski, What do we know about perspective-based reading? An approach for quantitative aggregation in software engineering, Presented at the Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, 2009.
39. A. Porter, L. Votta, Comparing detection methods for software requirements inspections: a replication using professional subjects, Empirical Softw. Eng. 3 (1998) 355–379.
40. A.A. Porter, L.G. Votta, An experiment to assess different defect detection methods for software requirements inspections, Presented at the Proceedings of the 16th international conference on Software engineering, Sorrento, Italy, 1994.
41. A.A. Porter, J. Lawrence, G. Votta, V.R. Basili, Comparing detection methods for software requirements inspections: a replicated experiment, IEEE Trans. Softw. Eng. 21 (June 1995) 563–575.
42. T. Thelin, P. Runeson, C. Wohlin, An experimental comparison of usage-based and checklist-based reading, IEEE Trans Softw. Eng. 29 (2003) 687–704.
43. O. Laitenberger, A survey of software inspection technologies, Handbook Softw. Eng. Knowl. Eng. 2 (2002) 517–555.
44. A. Aurum, H. Petersson, C. Wohlin, State-of-the-art: software inspections after 25 years, Softw. Test. Verif. Reliab. 12 (2002) 133–154.
45. G.H. Travassos, F. Shull, M. Fredericks, V.R. Basili, Detecting defects in object-oriented designs: using reading techniques to increase software quality, Presented at the Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, Colorado, United States, 1999.

46. Porter, A., Votta, L., Basili, V., “Comparing detection methods for software requirements inspections: a replicated experiment”, IEEE Transactions on Software Engineering, vol. 21, no. 6, pp.563-575, 1995.
47. Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., & Zelkowitz, M. V. (1996). The empirical investigation of perspective-based reading. Empirical Software Engineering, 1(2), 133-164.
48. X.M. Yang, Towards a Self-evolving Software Defect Detection Process, M.Sc. Thesis, Computer Science, University of Saskatchewan, Saskatoon, Saskatchewan, 2007.
49. G. Buckberry,and J. Siddiqi, proceedings of first international conference on Requirements Engineering, IEEE, 1994, pp.230-238
50. Ian Sommerville, Pete Sawyer, Requirements Engineering, John Wiley & Sons, New York, 1997.
51. Scientific Paper, “What is Requirements- Based Testing”, Author, Gary E. Mogyorodi, Published in, March 2003.
52. Beck, K. (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston, MA.
53. L. He, J. Carver, PBR vs. checklist: a replication in the n-fold inspection context, Presented at the Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering (ISESE’06), Rio de Janeiro, Brazil, 2006.
54. A. MacDonald, D. Russell, B. Atchison, Model-driven development within a legacy system: an industry experience report, Proc. Australian Software Engineering Conference, 2005, pp. 14-22.
55. Mohagheghi, P., Dehlen, V., & Neple, T. (2009). Definitions and approaches to model quality in model-based software development–A review of literature. *Information and software technology*, *51*(12), 1646-1669.
56. S.W. Ambler, The Elements of UML 2.0 Style, Cambridge University Press, 2005.
57. Kitchens, Tim. "Automating Software Development Processes." Published January 12 (2006): 2006.
58. Vyatkin, V. (2013). Software engineering in industrial automation: State-of-the-art review. IEEE Transactions on Industrial Informatics, 9(3), 1234-1249.
59. Piotr, S. (2009). Innovation in the software sector. OECD Publishing, pp 34-43.

60. Gupta, P., & Govil, M. C. (2010). MVC Design Pattern for the multi framework distributed applications using XML, spring and struts framework. International Journal on Computer Science and Engineering, 2(04), 1047-1051.
61. Dissanayake, N. R., & Dias, G. K. A. (2017). Balanced Abstract Web-MVC Style: An Abstract MVC Implementation for Web-based Applications. GSTF Journal on Computing, 5(3).
62. Brooks FJ. The Mythical Man-Month (anniversary edn). Addison-Wesley: Reading MA, 1995; 322 pp
63. Lehman MM. Software uncertainty and the role of CASE in its minimization and control. Proceedings of the 5th Jerusalem Conference on Information Technology. IEEE Computer Society Press: Los Alamitos CA, 1990.
64. Kemerer CF. An agenda for research in the managerial evaluation of computer-aided software engineering tool impacts. Proceedings of the 22nd Hawaii International Conference on System Sciences, vol. 2. IEEE Computer Society Press: Los Alamitos CA, 1989; 219–228. A
65. Lehman MM. Software uncertainty and the role of CASE in its minimization and control. Proceedings of the 5th Jerusalem Conference on Information Technology. IEEE Computer Society Press: Los Alamitos CA, 1990.
66. https://docs.microsoft.com/nl-be/visualstudio/modeling/generate-code-from-uml-class-diagrams?view=vs-2015. Accessed 21 June 2019.
67. Fuggetta, Alfonso, and Elisabetta Di Nitto. "Software process." Proceedings of the on Future of Software Engineering. ACM, 2014.
68. E. Dustin, J. Rashka, and J. Paul, Automated software testing: introduction, management, and performance
69. Karhu, K. Repo, T. Taipale, O. Smolander, K. Lappeenranta Univ. of Technol., Lappeenranta, Empirical Observations on Software Testing Automation. Software Testing Verification and Validation, 2009. ICST '09. International Conference on , vol,
70. E. Kit, Software Testing in the Real World: Improving the Process. Reading, MA: Addison-Wesley, 1995.J.
71. Hu Zhi-gen; Yuan Quan; Zhang Xi; , "Research on Agile Project Management with Scrum Method," Services Science, Management and Engineering, 2009. SSME '09. IITA

International Conference on , vol., no., pp.26-29, 11-12 July 2009 doi: 10.1109/SSME.2009.136

72. Collins Eliane, Lucena Vicente. 2010. Iterative Software Testing Process for Scrum and Waterfall Projects with Open Source Testing Tools Experience. In Proceedings of the 22nd IFIP International Conference on Testing Software and Systems2010(ICTSS'10). CRIM, 2010. 115-120 p. [ISBN : 978-2-89522-136-4] .
73. Crispin, L.; Gregory, J.; Agile Testing: A Practical Guide for Testers and Agile Teams, Addison-Wesley, 2009, ISBN 0-321-53446-8.
74. J. F. Smart. Jenkins The Definitive Guide. O'Reilly, 2011.
75. M. Fowler. Continous integration. http://martinfowler.com/articles/continuousIntegration.html,2006.
76. http://puppetlabs.com, Accessed 24 March 2017.
77. http://www.nagios.org, Accessed 17 April 2017.
78. P. Debois. Devops: A software revolution in the making? The Journal of Information Technology Management, 24(8):3–5, August 2001.
79. J. Humble and J. Molesky. Why enterprises must adopt devops to enable continous delivery. The Journal of Information Technology Management, 24(8):6–12, August 2001.
80. "Rational solution for systems and software engineering", http://pic.dhe.ibm.com/infocenter/rssehelp/v1r0m0/index.jsp?topic=%2Fcom.ibm.rational.sse.doc%2Ftopics%2Fc_overview_sse.html, http://www.youtube.com/watch?v=q1UwUkQZzW4&list=PLZGO0qYNSD4X757uGAmkWhieGknL4asHl&index=2, Accessed 10th June, 2014
81. "Requirements Engineering Suite (AcRES)", http://www.accenture.com/us-en/Pages/service-requirements-engineering-suite.aspx, Accessed 16th June, 2014
82. Young, Ralph R. "Recommended requirements gathering practices." CrossTalk 15, no. 4 (2002): 9-12.
83. Adeel, Kashif, Ahmad Shams and Akhtar Sohaib. "Defect prevention techniques and its usage in requirements gathering-industry practices." Engineering Sciences and Technology, 2005. SCONEST 2005. Student Conference on. IEEE, 2005. pp 1 – 5
84. A.R. Dennis, G.S. Hayes, R.M. Daniels Jr., Business process modeling with group support systems, Journal of Management Information Systems 15 (4), 1999, pp. 115–142.

85. Premkumar, G., & Potter, M. (1995). Adoption of computer aided software engineering (CASE) technology: an innovation adoption perspective. ACM SIGMIS Database: the DATABASE for Advances in Information Systems, 26(2-3), 105-124.
86. Bouchereau, V., & Rowlands, H. (2000). Methods and techniques to help quality function deployment (QFD). Benchmarking: An International Journal, 7(1), 8-20.
87. Mohd, Ibrahim and Rodina Ahmad. "Class diagram extraction from textual requirements using Natural language processing (NLP) techniques." Computer Research and Development, 2010 Second International Conference on Computer Research and Development. IEEE, 2010. pp 200-204.
88. Deeptimahanti, Deva Kumar and Ratna Sanyal. "Semi-automatic generation of UML models from natural language requirements." Proceedings of the 4th India Software Engineering Conference. ACM, 2011. pp 165-174.
89. Ambriola, V., & Gervasi, V. (1997, November). “Processing natural language requirements”. In Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference (pp. 36-45). IEEE.
90. Zhou, Xiaohua, and Nan Zhou. "Auto-generation of Class Diagram from Free-text Functional Specifications and Domain Ontology." (2008).
91. Chowdhury, Gobinda G. "Natural language processing." Annual review of information science and technology 37, no. 1 (2003): pp 51-89.
92. Ryan, K. “The role of natural language in requirements engineering”. Proceedings of the IEEE Int. Symposium on Requirements Engineering. San Diego, CA, pp. 240-242., (1993).
93. Osborne, Miles, and C. K. MacNish. "Processing natural language software requirement specifications." In Requirements Engineering, 1996., Proceedings of the Second International Conference on, pp. 229-236. IEEE, 1996.
94. Harmain, Harmain M., and R. Gaizauskas. "CM-Builder: an automated NL-based CASE tool." In Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on, pp. 45-53. IEEE, 2000.
95. Goldin, L., Berry, D.M.: “AbstFinder, a prototype natural language text abstraction finder for use in requirements elicitation”. Automated Software Eng. 4 (1997) pp.375–412

96. Kof, Leonid, "Natural language processing: mature enough for requirements documents analysis?" Natural Language Processing and Information Systems. Springer Berlin Heidelberg, 2005. pp 91-102.
97. Collins, M.: “Head-Driven Statistical Models for Natural Language Parsing”. PhD thesis, University of Pennsylvania (1999).
98. “Apache OpenNLP”: http://opennlp.sourceforge.net/Accessed 8th April, 2014
99. “WordNet A lexical database for English”, http://wordnet.princeton.edu/wordnet/ , Accessed 23rd April, 2014
100. Deeptimahanti, D. K. and Sanyal, R. “An Innovative Approach for Generating Static UML Models from Natural Language Requirements”. Springer Berlin Heidelberg, Communication in computer and Information Science, Advances in Software Engineering, Springer, Vol. 30, page 147 (2009)
101. Deva Kumar, D. and Sanyal, R. “Static UML Model Generator from Analysis of Requirements (SUGAR).” International Conference on Advanced Software Engineering and Its Applications (ASEA 2008), pp. 77-84 (2008)
102. Deeptimahanti, D. K. and Babar, M. A. “An Automated Tool for Generating UML Models from Natural Language Requirements”. IEEE / ACM Int. Conf. on ASE, 2009
103. “The Stanford Natural Language Processing Group”, http://nlp.stanford.edu/index.shtml Accessed 23rd April, 2014.
104. “JavaRAP”, http://aye.comp.nus.edu.sg/~qiu/NLPTools/JavaRAP.html, Accessed 23rd April, 2014.
105. “OMG XML Metadata Interchange”. Object Management Group, MOF 2.0/XMI Mapping, v.2.4.1, http://www.omg.org/spec/XMI/2.4.1/PDF/, Accessed 23rd April, 2014
106. Mich, L. NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA. Nat. Lang. Eng., 2, 2, pp. 161-187 (1996)
107. Börstler, J. User-Centered Requirements Engineering in RECORD - An Overview. Proc. of Nordic Workshop on Programming Environment Research’96, Denmark, pp. 149-156.
108. Nanduri, S. and Rugaber, S. Requirements validation via automated natural language parsing. Journal of Management Information Systems 1995-96; 12(3): pp. 9-19, 1996
109. Harmain, H. M. and Gaizauskas, R. CM-Builder: an automated NL-based CASE tool. In Proc. of the 15th IEEE Int. Conf. on Automated Software Engineering, pp. 45-53, (2000)

110. Overmyer, S. P., Benoit, L. and Owen, R. Conceptual modeling through linguistic analysis using LIDA. In Proc. of the 23rd Int. Conf. of Software Engineering (ICSE), Toronto, Canada, pp. 401–410, (2001)
111. Popescu, D., Rugaber, S., Medvidovic, N. and Berry, D. M. Reducing Ambiguities in Requirements Specifications Via Automatically Created Object-Oriented Models. In Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs. Springer, pp. 103-124. 2008.
112. Barry, E. J., Kemerer, C. F., & Slaughter, S. A. (2007). How software process automation affects software evolution: a longitudinal empirical analysis. Journal of Software Maintenance and Evolution: Research and Practice, 19(1), 1-31.
113. Kalaivani S, Dong L, Behrouz H. F and Eberlein, A. UCDA: Use Case Driven Development Assistant Tool for Class Model Generation. In Proc. of 16th Int. Conf. on Software Engineering and Knowledge Engineering, Banff, Canada, pp. 324-329, (2004).
114. Li, L. A Semi-Automatic Approach to Translating Use Cases to Sequence Diagrams. Proc. of the Technology of ObjectOriented Languages and Systems, pp.184, June 07-10, 1999.
115. Montes, A., Pacheco, H., Estrada, H. and Pastor, O. Conceptual Model Generation from Requirements Model: A Natural Language Processing Approach. LNCS. Vol. 5039 , pp. 325-326, Springer, 2008.
116. Isabel Diaz, Lidia Moreno, Inmaculada Fuentes and Pastor, O. Integrating Natural Language Techniques in OO-Methods. Computational Linguistics and Intelligent Text Processing, LNCS, Vol. 3406, pp. 177-188, Springer 2005.
117. Tao Yue, Lionel C Briand and Yvan Labiche, An Automated Approach to Transform Use Cases into Activity Diagrams, Modelling Foundations and Applications, LNCS, Volume 6138, pp. 337-353, (2010)
118. RAVENFLOW, http://www.ravenflow.com/, Accessed 6th June 2018.
119. Tweet NLP, http://www.cs.cmu.edu/~ark/TweetNLP/, accessed 13th October 2016.
120. Twitter Text Python, code repository, https://pypi.python.org/pypi/twitter-text-python/, accessed 13th October 2016.

121. Processing streaming data, https://dev.twitter.com/streaming/overview/processing, accessed 13th October 2016.
122. Khan, Mohammad, Markus Dickinson, and Sandra Kübler. "Does size matter? text and grammar revision for parsing social media data." Proceedings of the Workshop on Language Analysis in Social Media. 2013.
123. Natural Language Processing for Social Media, Volume 30 of Synthesis Lectures on Human Language Technologies, Atefeh Farzindar, Diana Inkpen, Morgan & Claypool Publishers, 2014.
124. "Part-of-Speech Tagging", https://www.cs.umd.edu/~nau/cmsc421/part-of-speech-tagging.pdf, Accessed 6th October, 2016.
125. Treebank, https://en.wikipedia.org/wiki/Treebank, accessed 13th October 2016.
126. Marcus, Mitchell P., Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a large annotated
127. Parse Tree, https://runestone.academy/runestone/books/published/pythonds/Trees/ParseTree.html, Accessed 18th November 2019.
128. Jackendoff, Ray, X Syntax. Cambridge, Mass.: MIT Press (1977).
129. Wen, W. (2012, June). Software fault localization based on program slicing spectrum. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 1511-1514). IEEE.
130. Gruber, Thomas R. "A translation approach to portable ontology specifications." *Knowledge acquisition* 5.2 (1993): 199-220.
131. Genesereth, Michael R., and Nils J. Nilsson. "Logical foundations of artificial. "*Intelligence. Morgan Kaufmann* (1987).
132. Oberle, D., Volz, R., Staab, S., & Motik, B. (2004). An extensible ontology software environment. In Handbook on ontologies (pp. 299-319). Springer, Berlin, Heidelberg.
133. De Nicola, A., Missikoff, M., & Navigli, R. (2009). A software engineering approach to ontology building. Information systems, 34(2), 258-275.

134. McGuinness, D. L., & Van Harmelen, F. (2004). OWL web ontology language overview. W3C recommendation, 10(10), 2004.
135. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P. F., & Rudolph, S. (2009). OWL 2 web ontology language primer. W3C recommendation, 27(1), 123.
136. "Ontology Development 101: A Guide to Creating Your First Ontology", http://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html, accessed 10th October 2016.
137. Protégé Wiki, https://protegewiki.stanford.edu/wiki/Main_Page, Accessed 18th November 2019.
138. Dou, D., Wang, H., & Liu, H. (2015, February). Semantic data mining: A survey of ontology-based approaches. In Proceedings of the 2015 IEEE 9th international conference on semantic computing (IEEE ICSC 2015) (pp. 244-251). IEEE.
139. Bechhofer, Sean. "OWL: Web ontology language." Encyclopedia of Database Systems. Springer US, 2009. 2008-2009.
140. B. Cuenca-Grau, I. Horrocks, B. Motik, B. Parsia, P.F. Patel-Schneider, U. Sattler, OWL 2: the next step for OWL, Journal of Web Semantics 6 (4) (2008) 309–322
141. W3C OWL Working Group, OWL 2 Web Ontology Language: Document Overview, 2009. <http://www.w3.org/TR/owl2-overview>. Accessed 23rd December 2016.
142. W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," Bull. Math. Biophys., vol. 5, pp. 115–133, 1943
143. N. Wiener, Cybernetics. Cambridge, MA: MIT Press, 1961.
144. Karayiannis, N., & Venetsanopoulos, A. N. (2013). Artificial neural networks: learning algorithms, performance evaluation, and applications (Vol. 209). Springer Science & Business Media. Pp 37-42.
145. Karayiannis, N., & Venetsanopoulos, A. N. (2013). Artificial neural networks: learning algorithms, performance evaluation, and applications (Vol. 209). Springer Science & Business Media. Pp 26 -36.
146. Schmidhuber, J. (2015). Deep learning in neural networks: An overview. Neural networks, 61, 85-117.

147. Gamboa, J. C. B. (2017). Deep learning for time-series analysis. arXiv preprint arXiv:1701.01887.
148. Sladojevic, S., Arsenovic, M., Anderla, A., Culibrk, D., & Stefanovic, D. (2016). Deep neural networks based recognition of plant diseases by leaf image classification. Computational intelligence and neuroscience, 2016.
149. Young, T., Hazarika, D., Poria, S., & Cambria, E. (2018). Recent trends in deep learning based natural language processing. ieee Computational intelligenCe magazine, 13(3), 55-75.
150. Stanford CoreNLP – Natural language software, https://stanfordnlp.github.io/CoreNLP/, Accessed 22nd November 2019.
151. Apache OpenNLP, https://opennlp.apache.org/, Accessed 22nd November 2019.
152. Dan Klein and Christopher D. Manning. 2003. Accurate Unlexicalized Parsing. Proceedings of the 41st Meeting of the Association for Computational Linguistics, pp. 423-430.
153. Anna Rafferty and Christopher D. Manning. 2008. Parsing Three German Treebanks: Lexicalized and Unlexicalized Baselines. In ACL Workshop on Parsing German.
154. Pi-Chuan Chang, Huihsin Tseng, Dan Jurafsky, and Christopher D. Manning. 2009. Discriminative Reordering with Chinese Grammatical Relations Features. In Proceedings of the Third Workshop on Syntax and Structure in Statistical Translation.
155. Spence Green and Christopher D. Manning. 2010. Better Arabic Parsing: Baselines, Evaluations, and Analysis. In COLING 2010.
156. Spence Green, Marie-Catherine de Marneffe, John Bauer, and Christopher D. Manning. 2010. Multiword Expression Identification with Tree Substitution Grammars: A Parsing tour de force with French.. In EMNLP 2011.
157. Most of the work on Spanish was by Jon Gauthier. There is no published paper, but you can thank him and/or cite this webpage: https://nlp.stanford.edu/software/spanish-faq.html, Accessed 25th November 2019.
158. Makoond, B., Elias, A., Talbot, S. R., Khaddaj, S., & Franczuk, S. (2014, August). ZDLC-Based Modelling and Simulation of Enterprise Systems. In 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace

Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS) (pp. 237-243). IEEE.

159. Fowler, M., Kobryn, C., & Scott, K. (2004). UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional.
160. Lange, C. F., Chaudron, M. R., & Muskens, J. (2006). In practice: UML software architecture and design description. IEEE software, 23(2), 40-46.
161. What is UML?, https://www.archimetric.com/what-is-uml/, Accessed 24th November 2019.
162. Cockburn, A. (1997). Structuring use cases with goals. Journal of Object-Oriented Programming, 10(5), 56-62.
163. Berardi, D., Calvanese, D., & De Giacomo, G. (2005). Reasoning on UML class diagrams. Artificial intelligence, 168(1-2), 70-118.
164. Routledge, N., Bird, L., & Goodchild, A. (2002). UML and XML schema. Australian Computer Science Communications, 24(2), 157-166.
165. Peak, R. S., Lubell, J., Srinivasan, V., & Waterbury, S. C. (2004). STEP, XML, and UML: complementary technologies. J. Comput. Inf. Sci. Eng., 4(4), 379-390.
166. https://www.omg.org/spec/XMI/About-XMI/
Accessed 6th November 2019.
167. Aho, A.V., Sethi, R. & Ullman, J.D. (1986). Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc. Boston, MA.
168. Yingxu, W., Yanan, Z., Philip, C., Xuhui, L. & Hong, G., (2010). The Formal Design Model of an Automatic Teller Machine (ATM). International Journal of Software Science and Computational Intelligence, 2(1): 102-131.
169. Yacim, J. A., & Boshoff, D. G. B. (2018). Impact of artificial neural networks training algorithms on accurate prediction of property values. *Journal of Real Estate Research*, *40*(3), 375-418.
170. https://towardsdatascience.com/power-of-a-single-neuron-perceptron-c418ba445095
Accessed 8th February 2021.